Robert C. **Martin**

CZYSTY KØD

PODRĘCZNIK DOBREGO PROGRAMISTY

Poznaj najlepsze metody tworzenia doskonałego kodu

Jak pisać dobry kod, a zły przekształcić w dobry?
Jak formatować kod, aby osiągnąć maksymalną czytelność?
Jak implementować pełną obsługę błędów bez zaśmiecania logiki kodu?

Helion

Tytuł oryginału: Clean Code: A Handbook of Agile Software Craftsmanship

Tłumaczenie: Paweł Gonera
Projekt okładki: Mateusz Obarek, Maciej Pokoński

ISBN: 978-83-283-1401-6

Authorized translation from the English language edition, entitled: Clean Code: A Handbook of Agile Software Craftsmanship, First Edition, ISBN 0132350882, by Robert C. Martin, published by Pearson Education, Inc., publishing as Prentice Hall.

Copyright © 2009 by Pearson Education, Inc.

Polish language edition published by Helion S.A.
Copyright © 2014.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Materiały graficzne na okładce zostały wykorzystane za zgodą iStockPhoto Inc.

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/czykov_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/czykov.zip

- Poleć książkę na Facebook.com
- Kup w wersji papierowej
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

SPIS TREŚCI

Słowo wstępne

JEDNYMI Z MOICH ULUBIONYCH DUŃSKICH CUKIERKÓW są Ga-Jol — ich silny zapach lukrecji jest doskonałym uzupełnieniem naszego wilgotnego i chłodnego klimatu. Częścią wizerunku Ga-Jol dla nas, Duńczyków, jest mądre lub dowcipne zdanie zapisane na górze każdego pudełka. Kupiłem dziś rano dwupak tego delikatesu i odkryłem, że jest na nim stare duńskie powiedzenie:

Ærlighed i små ting er ikke nogen lille ting.

„Uczciwość w małych rzeczach nie jest małą rzeczą". Był to dobry znak — właśnie o tym chciałem napisać we wprowadzeniu do niniejszej książki: małe rzeczy mają znaczenie. Jest to książka o niewielkich zagadnieniach, których znaczenie jest jednak niezwykle istotne.

„Bóg objawia się w szczegółach" — mówi architekt Ludwig Mies van der Rohe. Zdanie to przypomina współczesne argumenty na temat roli architektury w tworzeniu oprogramowania, a szczególnie w świecie Agile. Zdarzało się, że Bob i ja z pasją na ten temat dyskutowaliśmy. Faktycznie, Mies van der Rohe przykładał wielką wagę do użyteczności i ponadczasowej formy budynków, będących podstawą doskonałej architektury. Z drugiej strony, osobiście wybierał każdą klamkę w każdym projektowanym przez siebie domu. Dlaczego? Ponieważ małe rzeczy mają znaczenie.

W naszej „debacie" na temat TDD Bob i ja zgodziliśmy się co do tego, że architektura oprogramowania to jeden z najważniejszych aspektów jego tworzenia, ale chyba mamy inne wizje tego, co to dokładnie oznacza. Jednak takie wątpliwości są niezbyt istotne, ponieważ możemy przyjąć, że odpowiedzialni profesjonaliści przez *pewien* czas obmyślają i planują sposób realizacji projektu. Notacje z późnych lat dziewięćdziesiątych, w których wynikiem procesu projektowania były *wyłącznie* testy i kod, dawno już odeszły w zapomnienie. Jednak przywiązanie do szczegółów jest ważniejszą oznaką profesjonalizmu niż jakakolwiek wielka wizja. Po pierwsze, przez praktykę w niewielkiej

skali profesjonaliści zdobywają biegłość i zaufanie niezbędne w wielkiej skali projektów. Po drugie, najmniejszy fragment niewłaściwej konstrukcji — drzwi, które się nie domykają, nieco wybrzuszona deska w podłodze, a nawet bałagan na biurku — całkowicie burzą elegancję całości. To uzmysławia wagę czystego kodu w procesie projektowania aplikacji.

Nadal jednak architektura jest metaforą tworzenia oprogramowania, a w szczególności tej części programowania, która pozwala dostarczyć początkowy *produkt*, podobnie jak architekt dostarcza ogólny projekt budynku. W dzisiejszych czasach, gdy królują metodyki Scrum i Agile, priorytetem jest szybkie dostarczenie *produktu* na rynek. Chcemy, aby fabryka produkująca oprogramowanie działała z pełną wydajnością. Te „fabryki" stanowią koderzy, którzy pracują nad stworzeniem *produktu* od sporządzenia specyfikacji lub listy wymagań klienta. Metafora produkcyjna świetnie oddaje ten sposób myślenia — wszak sposób produkowania samochodów w japońskich fabrykach, w świecie linii produkcyjnych, zainspirował powstanie metodologii Scrum.

Jednak nawet w przemyśle samochodowym większość pracy nie stanowi produkowanie, ale utrzymanie pojazdów w stanie pełnej sprawności. W przypadku oprogramowania co najmniej 80% pracy można nazwać „utrzymaniem" — naprawianiem czegoś. Zamiast, jak typowy człowiek Zachodu, skupiać się na *produkowaniu* dobrego oprogramowania, powinniśmy myśleć raczej jak ktoś, kto remontuje budynki lub naprawia samochody. Co na *ten* temat mogą powiedzieć japońskie metody zarządzania?

Około 1951 roku w Japonii pojawiła się idea Totalnego Zarządzania Produkcją (Total Productive Maintenance — TPM). Skupiała się ona na utrzymaniu zamiast na produkcji. Jednym z głównych elementów TPM był zbiór tak zwanych zasad 5S. 5S to zbiór dyscyplin — i termin „dyscyplina" jest tu użyty w znaczeniu dosłownym. Te zasady 5S były w rzeczywistości podstawami metodyki Lean — coraz bardziej popularnego pojęcia w przemyśle oprogramowania. Zasady te nie są fakultatywne. Zgodnie z tym, co powiedział wujek Bob, praktyka dobrego oprogramowania wymaga takiej dyscypliny — skupienia, wyobraźni i myślenia. Nie zawsze jest to związane z działaniem, z nadawaniem taśmie produkcyjnej odpowiedniej prędkości. Filozofia 5S składa się z następujących koncepcji:

- *Seiri*, czyli organizacja. Koniecznie trzeba wiedzieć, jak znaleźć potrzebne rzeczy — przy użyciu odpowiednich narzędzi, na przykład nazewnictwa. Uważasz, że nazewnictwo nie jest ważne? Przeczytaj kolejne rozdziały.
- *Seiton*, czyli porządek. Istnieje stare amerykańskie powiedzenie: *Miejsce dla wszystkiego i wszystko na swoim miejscu*. Fragmenty kodu powinny znajdować się tam, gdzie się ich spodziewamy — jeżeli tak nie jest, powinniśmy przebudować kod, aby się tam znalazły.
- *Seiso*, czyli czystość. W miejscu pracy nie powinno być zwisających przewodów, piasku, ścinek ani śmieci. Tutaj autorzy mówią o zaśmiecaniu kodu komentarzami, które przechowują historię lub życzenia na przyszłość. Usuńmy je.
- *Seiketsu*, czyli standaryzacja. Grupa uzgadnia, w jaki sposób utrzymywać miejsce pracy w czystości. Czy uważasz, że w niniejszej książce znajdziesz coś na temat spójnego stylu kodowania i zbioru praktyk w zespole? Skąd biorą się te standardy? Zachęcam do lektury.
- *Shutsuke*, czyli dyscyplina (*samo*dyscyplina). Wymaga to zachowania dyscypliny w stosowaniu tych praktyk, częstego zwracania uwagi na swoją pracę i chęci do zmian.

Jeżeli podejmiesz wyzwanie — tak, wyzwanie — przeczytania tej książki i stosowania się do zamieszczonych w niej zaleceń, w końcu zrozumiesz i docenisz ostatni punkt. Docieramy wreszcie do podstaw pracy odpowiedzialnego profesjonalisty, który powinien być skupiony na cyklu życia produktu. Gdy utrzymujemy samochody lub inne maszyny zgodnie z zasadami TPM, to usuwanie awarii jest wyjątkiem. Zamiast tego wchodzimy na wyższy poziom — codziennie przeglądamy maszyny i naprawiamy zużywające się części, zanim się zepsują, wykonujemy wymianę oleju co 15 000 kilometrów itp. Kod przebudowujemy bez skrupułów. Możemy poprawić każdy z elementów. Inicjatywa TPM powstała 50 lat temu, by można było budować maszyny, które łatwiej jest serwisować. Zapewnienie czytelności kodu jest równie ważne jak zapewnienie jego działania. Kolejną z praktyk wdrożonych w ramach TPM około roku 1960 było skupienie się na wprowadzaniu nowych maszyn lub zastępowaniu starych. Jak poucza Fred Brooks, należy co siedem lat ponownie konstruować główne fragmenty oprogramowania, aby usunąć ujawnione błędy. Wydaje się, że powinniśmy skrócić stałą czasową Brooksa do tygodni, dni lub godzin, a nie lat. Tu właśnie nabierają znaczenia szczegóły.

W szczegółach tkwi siła, ale jest coś skromnego i głębokiego w takim podejściu do życia, coś, czego można się spodziewać w podejściu mającym japońskie korzenie. Jednak nie jest to tylko wschodnie spojrzenie na życie; mądrość ludowa ludzi Zachodu jest również pełna takich przestróg. Zasada *seiton*, wymieniona powyżej w jednym z punktów, wyszła spod pióra ministra z Ohio, który postrzegał schludność jako „lekarstwo na zło każdego rodzaju". Co można powiedzieć o *seiso*? *Czystość jest bliska boskości*. Niezależnie od tego, jak piękny jest dom, bałagan na biurku psuje dobre wrażenie. Co można znaleźć o *shutsuke* wśród złotych myśli? *Ten, kto jest godzien zaufania w sprawach niewielkich, jest godzien zaufania i w dużych*. Co możemy znaleźć na temat gotowości do przebudowy w odpowiednim czasie, na temat budowania fundamentów dla późniejszych „wielkich" decyzji, zamiast ich odkładania? *Jak sobie pościelisz, tak się wyśpisz*. *Nie odkładaj na jutro tego, co masz zrobić dzisiaj* (taki był oryginalny sens wyrażenia „ostatni odpowiedni moment" w Lean, gdy powiedzenie to wpadło w ucho konsultantom oprogramowania). Co można powiedzieć o małych, indywidualnych staraniach, podejmowanych podczas zmierzania do osiągnięcia wielkiego celu? *Wielki dąb rośnie z małego żołędzia*. Coś na temat wykonywania działań prewencyjnych w codziennym życiu? *Lepiej zapobiegać, niż leczyć*. *Jedno jabłko dziennie trzyma lekarza z daleka ode mnie*. Czysty kod jest zainspirowany mądrością naszej szerokiej kultury; kultury, w której wciąż jest obecna dbałość o szczegóły.

Nawet w wielkiej literaturze dotyczącej architektury znajdziemy symptomy skupienia się na szczegółach. Weźmy jako przykład klamki van der Rohe'a. To *seiri*. Dlatego powinniśmy zwracać uwagę na nazwę każdej zmiennej. Powinniśmy wybierać nazwy zmiennych z taką samą uwagą, jak wybieramy imię swojego dziecka.

Każdy właściciel domu wie, że konserwacja i ciągłe usprawnienia nie mają końca. Architekt Christopher Alexander — ojciec teorii wzorców w architekturze — postrzega każdy akt projektowania jako małą, lokalną naprawę. Uważa ponadto, że tworzenie precyzyjnej struktury jest jedynym zadaniem architekta; większe formy mogą być pozostawione wzorcom i ich stosowaniu przez mieszkańców. Projektowanie jest stałym zadaniem, nie tylko przy dodawaniu nowego pokoju do domu, ale również przy malowaniu, wymianie zniszczonych dywanów lub poprawianiu zlewu kuchennego. W sztuce pojawiają się analogiczne sentymenty. Szukając innych, którzy opisują Dom Boży jako pełen detali,

znaleźliśmy dobre towarzystwo w dziewiętnastowiecznym francuskim autorze, Gustawie Flaubercie. Francuski poeta, Paul Valéry, przekonuje, że poemat nigdy nie jest skończony i wymaga stałej pracy, a zaprzestanie pracy jest rezygnacją. Taka troska o szczegóły jest właściwa wszelkiemu dążeniu do doskonałości. Być może nie powiem nic nowego, ale będę w tej książce zachęcać do przyjęcia dyscypliny, która dawno temu została zastąpiona apatią lub tylko „odpowiadaniem na zmiany".

Niestety, zwykle nie uważamy takich problemów za kluczowe w programowaniu. Porzucamy nasz kod wcześnie nie dlatego, że jest gotowy, ale dlatego, że nasz system wartości skupia się na wyglądzie, a nie treści tego, co dostarczamy. Taki brak uwagi w końcu drogo nas kosztuje: *zły szeląg zawsze wraca*. Badania, zarówno w przemyśle, jak i na uniwersytetach, umniejszają wagę zachowania czystości kodu. Gdy pracowałem w Bell Labs Software Production Research, odkryliśmy, że spójny system wcięć był statystycznie największym wskaźnikiem niewielkiej liczby błędów. Chcieliśmy, aby tym wskaźnikiem jakości była architektura, język programowania albo inna zaawansowana notacja; jako osoby, które uważały za profesjonalizm doskonałość narzędzi i zaawansowane metody projektowania, czuliśmy się zdeprecjonowani przez wartość tych „maszyn" z najniższego poziomu fabryki, koderów, stosujących prosty i spójny styl wcięć. Odwołując się do mojej książki sprzed 17 lat, mogę powiedzieć, że taki styl oddzielał doskonałość od zwykłej kompetencji. Japończycy doceniają wartość poszczególnych pracowników i, co więcej, systemów produkcji, opierających się na prostych, codziennych działaniach tych pracowników. Jakość jest wynikiem milionów aktów uwagi — nie tylko wspaniałych metod spływających z niebios. To, że te działania są proste, nie oznacza, że są prostackie czy łatwe do wykonania. Są jednak bez wątpienia kanwą wielkości i piękna każdego przedsięwzięcia.

Oczywiście, niniejsza książka obejmuje szerszy zakres zagadnień, odwołując się do poglądów i doświadczeń prawdziwych pasjonatów dobrego kodu, takich jak Peter Sommerlad, Kevlin Henney oraz Giovanni Asproni. „Kod jest projektem" i „prosty kod" — to ich motta. Choć musimy pamiętać o tym, że interfejs jest programem i że struktura interfejsu wiele mówi o strukturze programu, niezwykle ważne jest odwoływanie się do zasady, która głosi, że projekt znajduje się w kodzie. Przebudowa w świecie produkcji wiąże się z kosztami, ale przebudowa projektu podwyższa jego wartość. Powinniśmy traktować nasz kod jako piękną artykulację poważnych zadań projektowych — projektu jako procesu, a nie statycznego punktu końcowego. To w kodzie znajdują się architektoniczne zasady łączenia i spójności. Jeżeli posłuchamy Larry'ego Constantine'a opisującego łączenie i spójność, to zauważymy, że mówi on z punktu widzenia kodowania, a nie wzniosłych koncepcji abstrakcyjnych, które można znaleźć w UML. Richard Gabriel przekonuje nas w swoim eseju *Abstraction Descant*, że abstrakcja to zło, chaos. Skoro tak, to kod, a szczególnie *czysty kod*, jest zaprzeczeniem zła, próbą uporządkowania chaosu.

Wracając do mojego małego pudełka Ga-Jol, myślę, że warto wspomnieć tu o duńskiej zasadzie, aby zwracać uwagę nie tylko na szczegóły, ale również być *uczciwym* w małych sprawach. Oznacza to, że powinniśmy być uczciwi w programowaniu, uczciwi względem kolegów, gdy mowa o stanie naszego kodu, a przede wszystkim uczciwi względem siebie i własnego kodu. Czy zrobiliśmy wszystko, aby „pozostawić obozowisko czyściejsze, niż je zastaliśmy"? Czy ulepszyliśmy nasz kod przed jego umieszczeniem w repozytorium? Nie są to zagadnienia peryferyjne — stanowią sedno zasad Agile. Metodologia Scrum zaleca, aby przebudowa kodu była częścią operacji zakończenia projektu. Ani architektura, ani czysty kod nie są doskonałe, a jedynie najlepsze, jakie mogliśmy

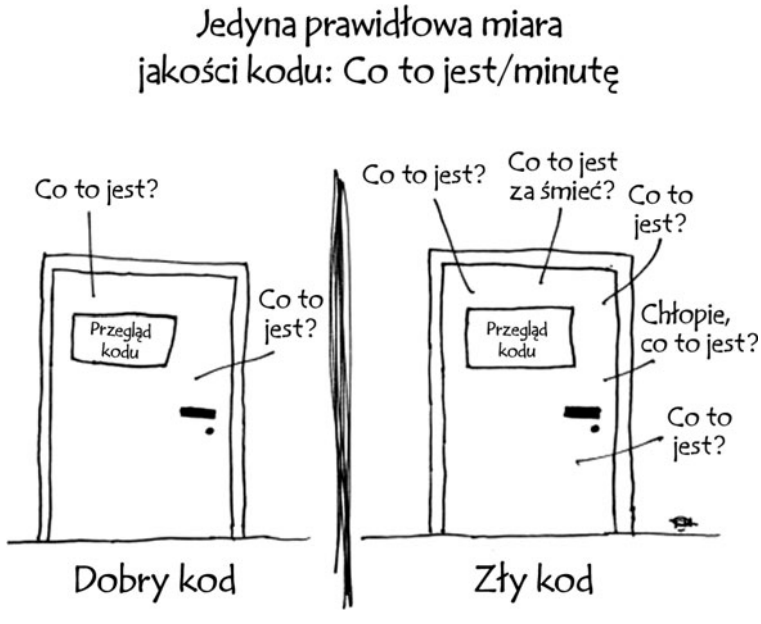
wykonać. *Mylić się jest rzeczą ludzką; wybaczać — boską*. W Scrum wszystko jest widoczne — niczego nie zamiatamy pod dywan. Jesteśmy uczciwi, jeśli chodzi o stan kodu, ponieważ kod nigdy nie jest doskonały.

W naszym zawodzie desperacko potrzebujemy całej pomocy, jaką możemy uzyskać. Jeżeli czysta podłoga w sklepie zmniejsza liczbę wypadków, a dobrze zorganizowany magazyn narzędzi zwiększa wydajność, to jestem za tym. Książka ta jest najlepszym pragmatycznym wdrożeniem zasad Lean w programowaniu, jakie kiedykolwiek widziałem na papierze. Tego oczekiwałem od tej małej grupy indywidualistów, którzy przez lata starali się nie tylko stać się lepszymi, ale również podarować swoją wiedzę przemysłowi, dzięki czemu teraz znajduje się ona w Twoich rękach. Pozostawiliśmy ten świat nieco lepszym, niż był przedtem, zanim wujek Bob przesłał mi rękopis.

Teraz, gdy napisałem te górnolotne stwierdzenia, zajmę się posprzątaniem biurka.

James O. Coplien
Mørdrup, Dania

Wstęp



Rysunek zamieszczony za zgodą Thoma Holwerda

Które drzwi reprezentują Twój kod? Które reprezentują Twój zespół lub firmę? Dlaczego jesteśmy w tym pokoju? Czy jest to zwykły przegląd kodu, czy może tuż po zainstalowaniu systemu pojawiło się mnóstwo poważnych problemów? Czy debugujemy w panice, ślęcząc nad kodem, nad którym wcześniej pracowaliśmy? Czy klienci odchodzą stadami, a kierownicy zabierają nam premie? Czy możemy być pewni, że znaleźliśmy się za *właściwymi* drzwiami, gdy sprawy się komplikują? Czy można jakoś nad tym zapanować? Odpowiedź jest prosta: należy osiągnąć *mistrzostwo*.

Dochodzenie do mistrzostwa wymaga dwóch elementów: wiedzy i pracy. Konieczne jest zdobywanie wiedzy na temat zasad, wzorców i heurystyk; następnie należy przekuć ją w umiejętności przez ciężką pracę i praktykę.

Możesz nauczyć się fizyki jazdy na rowerze. W rzeczywistości klasyczna matematyka jest względnie prosta. Grawitacja, tarcie, moment kątowy, środek masy i tak dalej — mogą być zademonstrowane na niecałej stronie równań. Mając te wzory, mogę udowodnić, że jazda na rowerze jest praktyczna, i dać Ci całą wiedzę, jaka jest potrzebna. Pomimo tego nadal będziesz spadał podczas pierwszych prób jazdy na rowerze.

Z kodowaniem jest podobnie. Moglibyśmy zapisać wszystkie „dobre" praktyki czystego kodu i zaufać Ci, że wykonasz swoją pracę (inaczej mówiąc, pozwolilibyśmy Ci spaść z roweru), ale jak by to świadczyło o nas jako nauczycielach i co dałoby Tobie jako uczniowi?

Nie. W tej książce zastosowaliśmy inne podejście.

Nauka pisania czystego kodu jest *ciężką pracą*. Wymaga czegoś więcej niż tylko wiedzy na temat zasad i wzorców. Musisz się przy tym *spocić*. Musisz to sam praktykować i przeżywać porażki. Musisz patrzeć na ćwiczenia innych i ich porażki. Musisz widzieć, jak dążą do doskonałości. Musisz widzieć, jak męczą się nad decyzjami i jaką cenę płacą, gdy decyzje te są złe.

Bądź przygotowany na ciężką pracę przy czytaniu tej książki. To nie jest książka „łatwa i przyjemna", którą można przeczytać w samolocie i skończyć przed lądowaniem. Będzie ona wymagała pracy, i to *ciężkiej*. Jakiego rodzaju będzie to praca? Będziesz czytał kod — wiele kodu. Będziesz proszony o zastanowienie się, co jest dobre, a co złe w danym kodzie. Będziesz proszony o śledzenie zmian podczas rozdzielania modułów na części i powtórnego ich składania. To będzie wymagało czasu i pracy, ale — zapewniamy — ta inwestycja zwróci się z nawiązką.

Książka została podzielona na trzy części. W kilku pierwszych rozdziałach opisujemy zasady, wzorce i praktyki pisania czystego kodu. W rozdziałach tych znajduje się sporo kodu i ich przeczytanie jest wyzwaniem. Przygotowują one do lektury drugiej części. Jeżeli odłożysz książkę po przeczytaniu pierwszej części, życzymy szczęścia!

Druga część książki wymaga cięższej pracy. Zawiera kilka analiz przypadków o coraz większej złożoności. Każda analiza przypadku jest ćwiczeniem polegającym na czyszczeniu pewnego kodu — poprawianiu kodu zawierającego różnej wagi usterki. Liczba szczegółów w tej części jest *ogromna*. Będziesz musiał skakać pomiędzy tekstem i listingami kodu, analizować tekst i śledzić nasze rozumowanie, aby pojąć każdą wprowadzoną zmianę. Zarezerwuj sobie trochę czasu, ponieważ powinno to zająć kilka dni.

Trzecia część książki jest nagrodą. Jest to jeden rozdział zawierający listę heurystyk i zapachów, jakie zebraliśmy przy tworzeniu analiz przypadków. Gdy analizowaliśmy i czyściliśmy kod studiów przypadków, dokumentowaliśmy każdy powód naszej akcji jako heurystykę lub zapach. Próbowaliśmy zrozumieć nasze reakcje dotyczące czytanego oraz zmienianego kodu i uchwycić przesłanki naszych założeń i decyzji. Wynikiem jest baza wiedzy opisująca sposób, w jaki myślimy, pisząc, czytając i czyszcząc kod.

Ta baza wiedzy ma ograniczoną wartość, jeżeli nie przeczytałeś uważnie analiz przypadków zawartych w drugiej części książki. W tych analizach uważnie odnotowaliśmy każdą zmianę w postaci odwołania do heurystyk. Te odwołania są umieszczone w nawiasach kwadratowych, np.: [H22].

Zwróć uwagę na kontekst, w jakim są stosowane i pisane te heurystyki. To nie same heurystyki są tak wartościowe, ale *relacje pomiędzy tymi heurystykami a konkretnymi decyzjami, jakie podjęliśmy w czasie czyszczenia kodu w analizie przypadku*.

Aby pokazać te relacje, umieściliśmy na końcu książki listę odwołań do heurystyk z numerami stron. Dzięki temu można znaleźć każde miejsce, w którym została zastosowana określona heurystyka.

Jeżeli przeczytasz pierwszą i trzecią część książki, pomijając analizy przypadków, będzie to lektura kolejnej „łatwej i przyjemnej" książki o pisaniu dobrego oprogramowania. Jeżeli jednak poświęcisz czas na pracę nad analizą przypadków, prześledzisz każdy mały krok, każdą decyzję — jeżeli postawisz się na naszym miejscu i przyjmiesz nasz sposób myślenia, to zdobędziesz znacznie lepszą wiedzę na temat tych zasad, wzorców, praktyk i heurystyk. Nie będziesz już potrzebował „łatwej i przyjemnej" wiedzy — stanie się ona częścią Twojej intuicji, częścią Ciebie, w taki sam sposób, jak rower staje się naszym przedłużeniem, gdy tylko opanujemy jazdę na nim.

Podziękowania

Rysunki

Dziękuję moim dwóm artystkom, Jeniffer Kohnke oraz Angeli Brooks. Jennifer jest autorką świetnych, pomysłowych rysunków zamieszczonych na początku każdego rozdziału oraz portretów Kenta Becka, Warda Cunninghama, Bjarne'a Stroustrupa, Rona Jeffriesa, Grady'ego Boocha, Dave'a Thomasa, Michaela Feathersa oraz mojej podobizny.

Angela jest autorką interesujących rysunków ozdabiających wnętrze każdego rozdziału. Narysowała dla mnie sporo rysunków przez ostatnie lata, w tym wiele ilustracji zamieszczonych w książce *Agile Software Development: Principles, Patterns, and Practices*. Angela jest moją najstarszą córką. Nie ukrywam, że jestem z niej bardzo dumny.

Czysty kod

Spodziewamy się, że Czytelnik sięgnął po tę książkę z dwóch powodów. Po pierwsze, jest programistą. Po drugie, chce być lepszym programistą. Dobrze. Potrzebujemy lepszych programistów.

Jest to książka o programowaniu, dlatego zamieszczono w niej sporo kodu. Przyglądamy się temu kodowi bardzo wnikliwie i z różnych stron, o czym przekonasz się w trakcie czytania. Dzięki temu będziesz dostrzegał różnicę pomiędzy dobrym kodem a złym. Wiemy, jak pisać dobry kod. Wiemy również, jak naprawić kiepski kod.

Niech stanie się kod...

Wielu Czytelników uważa zapewne, że książka na temat kodu jest anachronizmem — wszak kod nie stanowi dziś problemu; powinniśmy skupić się na modelach i wymaganiach. Faktycznie, niektóre osoby sugerowały, że jesteśmy blisko końca epoki kodu. Że wkrótce cały kod będzie generowany, a nie pisany. Że programiści po prostu nie będą potrzebni, ponieważ specjaliści biznesowi będą generowali program na podstawie specyfikacji.

Nonsens! Nigdy nie unikniemy stosowania kodu, ponieważ implementuje on szczegółowe wymagania użytkowników. Na pewnym poziomie pracy nad programem szczegóły te nie mogą być zignorowane — trzeba dokładnie je określić. Specyfikacja wymagań na takim poziomie szczegółowości, jaki maszyna może wykonać, *jest właśnie programowaniem*. Specyfikacja ta *jest kodem*.

Oczekuję, że poziom abstrakcji naszych języków będzie stale wzrastał, podobnie jak liczba języków specyficznych dla domeny. Jest to korzystne zjawisko, jednak nie wyeliminujemy kodu. W rzeczywistości wszystkie specyfikacje napisane w tych językach wyższego poziomu oraz właściwych dla domeny *będą* kodem! Będą one nadal musiały być rygorystyczne, dokładne i tak formalne i szczegółowe, aby maszyna mogła je zinterpretować i wykonać.

Osoby, które uważają, że kod pewnego dnia zniknie, są podobne do matematyków, którzy mają nadzieję, że pewnego dnia odkryją matematykę... niesformalizowaną. Mają oni nadzieję, że pewnego dnia stworzą maszyny, które będą mogły wykonywać to, co będziemy chcieli, a nie to, co im nakażemy. Maszyny te będą potrafiły przekształcać ogólnie wyrażone potrzeby na doskonale działające programy, które precyzyjnie spełnią nasze wymagania.

To się nigdy nie zdarzy. Nawet ludzie, ze swoją intuicją i pomysłowością, nie są w stanie tworzyć udanych systemów na podstawie niejasnych odczuć swoich klientów. Jeżeli dyscyplina specyfikacji wymagań czegokolwiek nas nauczyła, to właśnie konieczności tworzenia precyzyjnie określonych wymagań, które są tak formalne jak kod i mogą służyć jako testy wykonania tego kodu!

Należy pamiętać, że kod jest w rzeczywistości językiem, w którym ostatecznie wyrażamy wymagania. Możemy tworzyć języki, które są bliskie wymaganiom. Możemy tworzyć narzędzia, które pomagają nam analizować i przetwarzać wymagania na struktury formalne. Jednak nigdy nie wyeliminujemy niezbędnej precyzji — przyszłość kodu jest zatem niezagrożona.

W poszukiwaniu doskonałego kodu...

Ostatnio czytałem przedmowę do książki Kenta Becka *Implementation Patterns*[1]. Kent napisał: „Książka ta bazuje na dość kruchych podstawach — na tym, że dobry kod ma znaczenie". *Kruche* podstawy? Nie zgadzam się z tym! Uważam, że jest to jedna z najistotniejszych przesłanek w programowaniu (i myślę, że Kent o tym wie). Wiemy, że dobry kod ma znaczenie, ponieważ tak długo musieliśmy zmagać się z jego brakiem.

---

[1] [Beck07].

Więcej na: www.ebook4all.pl

Znam pewną firmę, która pod koniec lat osiemdziesiątych ubiegłego wieku opracowała *niesamowitą* aplikację. Była ona bardzo popularna i wielu profesjonalistów ją kupiło. Jednak wkrótce cykle produkcyjne zaczęły się wydłużać. Błędy z jednej wersji nie były poprawiane w następnej. Czas uruchamiania aplikacji wydłużał się, awarie pojawiały się coraz częściej. Pamiętam, że pewnego dnia po prostu wyłączyłem ten program i nigdy więcej go nie użyłem. Niedługo później firma ta wypadła z rynku.

Dwadzieścia lat później spotkałem jednego z pracowników tej firmy i zapytałem go, co się stało. Odpowiedź potwierdziła moje obawy. Śpieszyli się z produktem i mieli ogromny bałagan w kodzie. Dodawali coraz więcej funkcji, a kod stawał się coraz gorszy i gorszy, aż doszło do tego, że po prostu nie dało się nim zarządzać. *Był to przypadek zniszczenia firmy przez zły kod.*

Czy *Czytelnikowi* zdarzyło się być wstrzymywanym przez zły kod? Każdy programista o dowolnym doświadczeniu spotkał się wielokrotnie z takim przypadkiem. Mamy nawet na to nazwę. Jest to tzw. *brodzenie* (ang. *wading*). Brodzimy w złym kodzie. Przedzieramy się przez bajoro pokręconych gałęzi i ukrytych pułapek. Staramy się znaleźć drogę naprzód, mając nadzieję na jakąś podpowiedź, jakiś pomysł na to, co się dzieje, ale rozwiązanie niknie w bezmiarze bezsensu.

Oczywiście, każdy z nas stracił mnóstwo czasu z powodu złego kodu. A zatem — dlaczego go tworzymy?

Czy wynika to z pośpiechu? Prawdopodobnie tak. Często uważamy, że nie mamy czasu, aby dobrze wykonać swoją pracę; że nasz szef będzie zły, jeżeli będziemy spędzali czas na usprawnianiu kodu. Być może jesteśmy już zmęczeni pracą nad programem i chcemy ją zakończyć. Zdarza się również, że zaglądamy do listy zadań, jakie obiecaliśmy wykonać, i zauważamy, że trzeba szybko sklecić jakiś moduł, aby przejść do następnego. Wszyscy tak robimy.

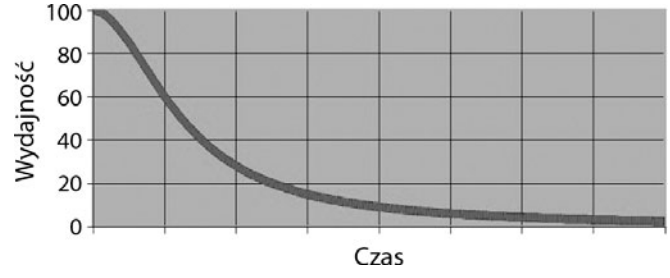
Nie zaglądamy do bałaganu, jaki właśnie zrobiliśmy, i zostawiamy to na inny dzień. Uważamy, że najważniejsze jest, że nasz zabałaganiony program działa, i uznajemy, że działający bałagan jest lepszy niż nic. Obiecujemy sobie, że kiedyś do niego wrócimy i go poprawimy. Oczywiście, znamy prawo LeBlanca: *Później znaczy nigdy*.

Całkowity koszt bałaganu

Każdy programista z kilkuletnim doświadczeniem w kodowaniu z pewnością stracił wiele czasu pracy, borykając się z czyimś niechlujnym kodem. Zwykle jest on przyczyną znaczącego spowolnienia pracy. W czasie roku lub dwóch zespoły, które działały bardzo szybko na początku projektu, zauważają, że posuwają się w żółwim tempie. Każda zmiana wprowadzona do kodu powoduje błędy w dwóch lub trzech innych fragmentach kodu. Żadna zmiana nie jest łatwa. Każdy dodatek lub modyfikacja systemu wymaga „zrozumienia" skrzyżowań, zakrętów i węzłów, aby można

było dodać kolejne skrzyżowania, zakręty i węzły. Z czasem bałagan staje się tak potężny, że nie można się go pozbyć. Po prosu nie ma na to sposobu.

Wraz ze wzrostem bałaganu efektywność zespołu stale spada, dążąc do całkowitego paraliżu projektu. Wobec spadku efektywności kierownictwo zwykle dodaje do projektu więcej osób w nadziei, że podniesie to wydajność zespołu. Jednak nowe osoby nie są zaznajomione z projektem systemu. Nie wiedzą, na jakie zmiany projektu mogą sobie pozwolić, a jakie spowodują jego naruszenie. Dodatkowo, tak jak wszyscy pozostali w zespole, są pod ogromną presją zwiększenia wydajności. Tak więc tworzą kolejne pokłady bałaganu, spychając wydajność jeszcze bliżej zera (patrz rysunek 1.1).



RYSUNEK 1.1. Wydajność a czas

Rozpoczęcie wielkiej zmiany projektu

W końcu zespół buntuje się. Informuje kierownictwo, że nie może kontynuować programowania na bazie tego okropnego kodu. Żąda przeprojektowania. Kierownictwo nie chce kierować wszystkich zasobów do pełnej zmiany projektu, ale nie może zaprzeczyć, że wydajność zespołu jest fatalna. W końcu ugina się pod żądaniami programistów i zatwierdza wielką zmianę projektu.

Wybierany jest nowy zespół tygrysów. Każdy chce być w tym zespole, ponieważ jest to projekt tworzony od początku. Ludzie chcą napisać coś naprawdę pięknego. Jednak do tego zespołu są wybierani tylko najlepsi i najbardziej błyskotliwi. Reszta musi nadal utrzymywać bieżący system.

Teraz te dwa zespoły zaczynają się ze sobą ścigać. Zespół tygrysów musi zbudować nowy system, który realizuje wszystkie funkcje starego. Oprócz tego musi również uwzględniać zmiany wprowadzane do starego systemu. Kierownictwo nie zdecyduje się na wymianę starego systemu, dopóki nowy nie będzie realizował tego, co poprzedni.

Wyścig będzie trwał bardzo długo. Znam przypadek, gdy trwało to 10 lat. Po tym czasie pierwsi członkowie zespołu tygrysów dawno odeszli, a pozostali zatrudnieni żądali przeprojektowania nowego systemu, ponieważ był on bardzo zabałaganiony.

Jeżeli ktoś zetknął się z tym zjawiskiem, wie już, że utrzymywanie wysokiej jakości kodu jest nie tylko efektywne finansowo, ale także jest warunkiem przetrwania firmy na rynku.

Postawa

Czy zdarzyło się Czytelnikowi przedzierać przez taki gąszcz kodu, że zmiana, która wymagała godzin, zajęła tygodnie? Czy zdarzyło się, że coś, co powinno być zmianą w jednym wierszu, spowodowało zmianę setek wierszy w różnych modułach? Objawy te występują nazbyt często.

Dlaczego tak się dzieje? Dlaczego dobry kod tak szybko się psuje? Mamy na to wiele wyjaśnień. Narzekamy, że zbyt daleko idące zmiany wymagań zniweczyły początkowy projekt. Stwierdzamy, że plany były zbyt napięte, aby można było wykonać wszystko prawidłowo. Mówimy o głupich kierownikach i nietolerancyjnych klientach, bezużytecznych marketingowcach i sesjach telefonicznych. Jednak wina, drogi Dilbercie, jest nie w gwiazdach, ale w nas samych — w braku profesjonalizmu.

Być może jest to gorzka pigułka do przełknięcia. Jak to *nasza* wina? A wymagania? Co z harmonogramem? Co z głupimi kierownikami i bezużytecznymi marketingowcami? Czy nie są oni po części winni?

Nie. Kierownictwo i marketing u *nas* szuka informacji, jakich potrzebuje, aby złożyć obietnice i przeprowadzić uzgodnienia, a jeśli nawet o nic nas nie pytają, nie powinniśmy się wstydzić powiedzieć, co myślimy. Użytkownicy oczekują od nas sprawdzenia, czy wymagania będą pasować do systemu. Kierownicy projektu oczekują, abyśmy pomogli im w planowaniu. Powinniśmy być zainteresowani planowaniem projektu i brać znaczną część odpowiedzialności za wszystkie błędy, szczególnie jeżeli błędy te są związane ze złym kodem!

Powiesz: „Zaraz, zaraz. Jeżeli nie zrobię tego, co mi kazał mój kierownik, zostanę zwolniony". Prawdopodobnie nie. Większość kierowników to ludzie, którzy chcą prawdy, nawet jeżeli tak się nie zachowują; chcą dobrego kodu, nawet jeżeli mają obsesję na punkcie harmonogramów. Mogą z pasją bronić harmonogramów i wymagań — to ich praca. *Naszą* pracą jest obrona kodu z równie wielką pasją.

Wyobraźmy sobie następującą sytuację: przed operacją pacjent zdecydowanie żąda, aby lekarz przestał wreszcie myć ręce, ponieważ zabiera to zbyt wiele czasu[2]. Jasne, że pacjent jest szefem, ale każdy lekarz absolutnie odrzuci takie żądania. Dlaczego? Ponieważ lekarze wiedzą więcej niż pacjent na temat ryzyka choroby i infekcji. Byłoby to nieprofesjonalne (a oprócz tego karalne), gdyby lekarz w tym przypadku zgodził się z pacjentem.

Tak więc brakiem profesjonalizmu programisty jest podporządkowanie się kierownikowi, który nie rozumie zagrożeń związanych z bałaganem w kodzie.

---

[2] W roku 1847 Ignaz Semmelweis zalecał lekarzom mycie rąk, ale oni nie stosowali się do tego, twierdząc, że są zbyt zajęci i nie mają na to czasu pomiędzy wizytami u kolejnych pacjentów.

Największa zagadka

Programiści spotykają się z zagadką podstawowych wartości. Wszyscy programiści mający co najmniej kilka lat doświadczenia wiedzą, że bałagan ich spowalnia. Jednocześnie wszyscy programiści ulegają presji tworzenia bałaganu w celu dotrzymania terminów. W skrócie — nie uznają, że czas biegnie tak szybko!

Prawdziwi profesjonaliści wiedzą, że druga część zagadki jest niewłaściwa. *Nie da* się dotrzymać terminów, wprowadzając bałagan. W rzeczywistości bałagan natychmiast nas spowolni, co spowoduje przekroczenie terminu. *Jedynym* sposobem na dotrzymanie terminu — jedynym sposobem na szybki rozwój aplikacji — jest utrzymywanie możliwie najwyższej czystości kodu.

Sztuka czystego kodu?

Załóżmy, że uważasz niedbały kod za znaczne utrudnienie. Załóżmy, że akceptujesz fakt, iż jedynym sposobem na utrzymanie dużej szybkości tworzenia kodu jest zapewnienie jego czystości. Trzeba sobie zadać pytanie: „Czy piszę czysty kod?". Próba tworzenia czystego kodu nie ma większego sensu, jeżeli nie wiadomo, co to stwierdzenie oznacza!

Tworzenie czystego kodu można porównać do malowania obrazu. Większość z nas może stwierdzić, czy obraz jest ładny czy brzydki. Jednak możliwość odróżnienia dobrego dzieła od złego nie oznacza, że wiemy, jak malować. Podobnie zdolność do odróżnienia czystego kodu od niedbałego nie oznacza, że wiemy, jak pisać czysty kod!

Pisanie takiego kodu wymaga dyscypliny oraz wykorzystania wielu technik w kolejnych mozolnych iteracjach pozwalających na osiągnięcie „czystości". Takie „czucie kodu" jest kluczem do sukcesu. Niektórzy z nas po prostu się z tym urodzili. Niektórzy musieli tę umiejętność zdobyć. Zdolność ta nie tylko pozwala stwierdzić, czy kod jest dobry, czy zły, ale również pokazuje strategie stosowania technik naszej dyscypliny nauki do przekształcenia złego kodu w czysty.

Programista pozbawiony „czucia kodu" może patrzeć na bałagan i rozpoznawać go, ale nie będzie miał pomysłu, co z nim zrobić. Programista *posiadający* tę umiejętność spojrzy na zabałaganiony moduł i zobaczy możliwości i warianty. Takie „czucie kodu" pomaga programiście wybrać najlepszy wariant i poprowadzi go (lub ją) do naszkicowania sekwencji działań zapewniających transformację kodu z postaci bieżącej do finalnej.

W skrócie — programista piszący czysty kod jest artystą, który po serii transformacji przekształca pusty ekran edytora w elegancko zakodowany system.

Co to jest czysty kod?

Istnieje prawdopodobnie tyle definicji, ilu jest programistów. Dlatego poprosiłem kilku znanych i bardzo doświadczonych programistów o opinie na ten temat.

Bjarne Stroustrup, twórca C++ oraz autor *The C++ Programming Language*

Lubię, gdy mój kod jest elegancki i efektywny. Logika kodu powinna być prosta, aby nie mogły się w niej kryć błędy, zależności minimalne dla uproszczenia utrzymania, obsługa błędów kompletna zgodnie ze zdefiniowaną strategią, a wydajność zbliżona do optymalnej, aby nikogo nie kusiło psucie kodu w celu wprowadzenia niepotrzebnych optymalizacji. Czysty kod wykonuje dobrze jedną operację.

Bjarne użył słowa „elegancki". To właściwe słowo! Słownik języka polskiego podaje następującą definicję słowa „elegancja": *dobry smak i wytworność w sposobie bycia; przyjemnie kunsztowny i prosty*. Warto zwrócić uwagę na słowo „przyjemny". Wydaje się, że Bjarne uważa czysty kod za *przyjemny* w czytaniu. Czytanie go powinno wywoływać uśmiech na twarzy, podobnie jak dobrze skomponowana muzyka czy dobrze zaprojektowany samochód.

Bjarne wspomina również o efektywności — i to *dwukrotnie*. Ten pogląd twórcy języka C++ nie powinien nas dziwić, ale osobiście uważam, że jest to coś więcej niż tylko zwykła potrzeba szybkości działania. Następne warte do odnotowania stwierdzenie opisuje konsekwencje braku elegancji. Użył on słowa „kusić". Jest to szczera prawda. Zły kod *kusi* do powiększenia bałaganu! Zmiany przeprowadzane w złym kodzie zwykle prowadzą do tego, że staje się on jeszcze gorszy.

Pragmatyczni Dave Thomas oraz Andy Hunt powiedzieli to w inny sposób. Użyli oni metafory rozbitego okna[3]. Budynek z rozbitymi oknami wygląda, jakby nikt się o niego nie troszczył. Dlatego inni ludzie również przestają się o niego troszczyć. W ten sposób coraz więcej okien jest rozbijanych. W końcu sami je rozbijają. Zaczynają bazgrać na fasadzie i pozwalają na powstanie gór śmieci. Jedno rozbite okno rozpoczyna proces upadku.

Bjarne wspomniał również, że obsługa błędów powinna być kompletna. Wymaga to skupienia się na szczegółach. Skrócona obsługa błędów jest jednym z przypadków, gdy programiści nie dbają o szczegóły. Drugim są wycieki pamięci, a trzecim wyścigi. Kolejnym takim przypadkiem są niespójne nazwy. Wynika z tego, że w czystym kodzie zwracamy uwagę na szczegóły.

Bjarne zamyka swoją wypowiedź stwierdzeniem, że czysty kod wykonuje dobrze jedną operację. Nie jest przypadkiem, że tak wiele zasad projektowania oprogramowania można sprowadzić do jednego prostego stwierdzenia. W złym kodzie próbuje się wykonać zbyt wiele zadań i operacji, mieszając zamierzenia i ambicję z zastosowaniem. Czysty kod jest *ukierunkowany*. Każda funkcja, każda klasa, każdy moduł realizuje jedno zadanie.

---

[3] *http://www.pragmaticprogrammer.com/booksellers/2004-12.html*

Grady Booch, autor *Object Oriented Analysis and Design with Applications*

> *Czysty kod jest prosty i bezpośredni. Czysty kod czyta się jak dobrze napisaną prozę. Czysty kod nigdy nie zaciemnia zamiarów projektanta; jest pełen trafnych abstrakcji i prostych ścieżek sterowania.*

Grady powtórzył niektóre spostrzeżenia Bjarne, ale ujął je z perspektywy *czytelności*. Moim zdaniem bardzo trafne jest stwierdzenie, że czysty kod powinien być podobny do dobrze napisanej prozy. Przypomnijmy sobie naprawdę dobrą książkę, jaką ostatnio czytaliśmy — słowa płynnie wytwarzały obrazy w naszych myślach. Było to podobne do oglądania filmu, prawda? Lepiej! Widzieliśmy bohaterów, słyszeliśmy dźwięki, doświadczaliśmy patosu i humoru.

Jednak czytanie czystego kodu nie będzie podobne do czytania *Władcy pierścieni*, mimo że metafora literacka jest dość adekwatna. Podobnie jak w przypadku dobrej powieści, czysty kod jasno przedstawia napięcie w rozwiązywanym problemie. Powinien on budować to napięcie, aż do punktu kulminacyjnego, w którym czytelnik krzyknie: „Aha! Oczywiście!", gdy pojawi się oczywiste rozwiązanie.

Zauważyłem, że Grady używa stwierdzenia „trafna abstrakcja" jako fascynującego oksymoronu! W tym znaczeniu słowo „trafna" jest niemal synonimem „konkretna". Mój słownik definiuje słowo „trafna" jako *energicznie stanowcza, rzeczowa, pozbawiona niezdecydowania i niepotrzebnych szczegółów*. Pomimo tych pozornie różnych znaczeń słowo to niesie ze sobą mocny przekaz. Nasz kod powinien być rzeczowy, a nie spekulacyjny. Powinien zawierać tylko to, co jest niezbędne. Nasi czytelnicy powinni postrzegać nas jako skutecznych.

„Big" Dave Thomas, założyciel OTI, ojciec chrzestny strategii Eclipse

> *Czysty kod może być czytany i rozszerzany przez innego programistę niż jego autor. Posiada on testy jednostkowe i akceptacyjne. Zawiera znaczące nazwy. Oferuje jedną, a nie wiele ścieżek wykonania jednej operacji. Posiada minimalne zależności, które są jawnie zdefiniowane, jak również zapewnia jasne i minimalne API. Kod powinien być opisywany przy jednoczesnej zależności od języka — nie wszystkie potrzebne informacje mogą być wyrażane bezpośrednio w kodzie.*

„Big" Dave również wskazuje potrzebę czytelności przedstawioną przez Grady'ego, ale z ważną różnicą. Dave zakłada, że czysty kod pozwala na jego łatwe rozszerzanie przez *innych* ludzi. Może to się wydawać oczywiste, ale nie może być niedocenione. Jest to w końcu różnica pomiędzy kodem, jaki się łatwo czyta, a kodem, jaki można łatwo zmieniać.

Dave łączy czystość z testami! Dziesięć lat temu spowodowałoby to powszechne zdziwienie. Jednak dyscyplina programowania sterowanego testami w znacznym stopniu wpłynęła na nasz przemysł i stała się jedną z najbardziej podstawowych zasad. Dave ma rację. Kod bez testów nie jest czysty. Nie ma znaczenia, jak jest elegancki, nie ma znaczenia, jak jest czytelny i dostępny; jeżeli brakuje w nim testów, nie jest czysty.

Dave dwukrotnie używa słowa *minimalny*. Najwyraźniej docenia potrzebę minimalizowania kodu. Faktycznie, jest to często spotykane twierdzenie w literaturze dotyczącej programowania. Mniejsze jest lepsze.

Dave uważa również, że kod powinien być *opisywany*. Jest to odwołanie do *programowania piśmiennego*[4] Knutha. Wynikiem tego procesu powinien być kod skomponowany w taki sposób, aby był czytelny dla ludzi.

Michael Feathers, autor *Working Effectively with Legacy Code*

> *Mógłbym wymieniać wszystkie cechy, jakie zauważam w czystym kodzie, ale istnieje jedna, która prowadzi do pozostałych. Czysty kod zawsze wygląda, jakby był napisany przez kogoś, komu na nim zależy. Nie ma w nim nic oczywistego, co mógłbyś poprawić. Autor kodu pomyślał o wszystkim i jeżeli próbujemy sobie wyobrazić usprawnienia, prowadzą one nas tam, skąd zaczęliśmy, co pozwala nam docenić kod, który ktoś dla nas napisał — kod, który napisał ktoś, kto naprawdę przejmuje się swoimi zadaniami.*

Jedno wyrażenie: „przejmuje się". Tak naprawdę jest to temat tej książki. Prawdopodobnie odpowiednim podtytułem byłoby zdanie *Jak przejmować się kodowaniem*.

Michael trafił w sedno. Czysty kod to taki, którym zajął się ktoś, komu na nim zależy. Ktoś, kto poświęcił czas na jego uproszczenie i uporządkowanie. Ktoś, kto zwrócił uwagę na szczegóły — kto się *przejął*.

---

[4] [Knuth92].

Ron Jeffries, autor *Extreme Programming Installed* oraz *Extreme Programming Adventures in C#*

Ron zaczął swoją karierę od programowania w języku Fortran w Strategic Air Command i pisał kod w niemal wszystkich językach i na niemal każdej maszynie. Warto uważnie zapoznać się z jego słowami.

> *Ostatnio zacząłem (i niemal skończyłem) od zasad Becka dotyczących prostego kodu. W podanej kolejności prosty kod:*
>
> - *przechodzi wszystkie testy;*
> - *nie zawiera powtórzeń;*
> - *wyraża wszystkie idee projektowe zastosowane w systemie;*
> - *minimalizuje liczbę encji, takich jak klasy, metody, funkcje i podobne.*
>
> *Spośród wymienionych cech skupiam się głównie na powtórzeniach. Gdy ta sama operacja jest wykonywana wielokrotnie, jest to znak, że znajduje się tam idea, która nie jest wystarczająco dobrze reprezentowana w kodzie. Próbuję określić, co nią jest. Następnie próbuję wyrazić tę ideę znacznie jaśniej.*
>
> *Ekspresywność oznacza dla mnie użycie znaczących nazw i osobiście zmieniam kilkakrotnie nazwy różnych elementów, zanim znajdę właściwą. Przy użyciu nowoczesnych narzędzi, takich jak Eclipse, zamiana nazwy jest dosyć prosta, więc nie stanowi to dla mnie problemu. Ekspresywność nie ogranicza się jednak do nazw. Sprawdzam również, czy obiekt lub metoda nie wykonują więcej niż jednej operacji. Jeżeli jest to obiekt, prawdopodobnie wymaga podziału na dwa lub więcej obiektów. Jeżeli jest to metoda, zawsze korzystam w niej z narzędzia Extract Method, dzięki czemu uzyskuję jedną metodę, która jaśniej wyraża to, co wykonuje, i kilka podmetod mówiących, jak jest to zrobione.*
>
> *Powtórzenia i ekspresywność są początkiem bardzo długiej drogi do tego, co uważam za czysty kod, a usprawnienie brudnego kodu pod względem tylko tych dwóch cech powoduje znaczną różnicę. Istnieje jednak jeszcze jedna rzecz, na jaką zwracam uwagę; jest ona nieco trudniejsza do wyjaśnienia.*
>
> *Po latach wykonywania tej pracy wydaje mi się, że wszystkie programy składają się z bardzo podobnych elementów. Jednym z przykładów jest „szukanie elementów w kolekcji". Niezależnie od tego, czy mamy bazę danych z informacjami o pracownikach, czy tablicę mieszającą z kluczami i wartościami, czy też tablicę z pewnego rodzaju elementami, często chcemy uzyskać określony element z takiej kolekcji. Gdy znajdę taki fragment, często przenoszę określoną implementację do bardziej abstrakcyjnej metody lub klasy. Daje mi to kilka interesujących możliwości.*
>
> *Mogę teraz zaimplementować funkcję przy użyciu prostego mechanizmu, na przykład tablicy mechanizmu, ale ponieważ w tym momencie wszystkie odwołania do tego wyszukiwania odwołują się do mojej małej abstrakcji, mogę zmienić implementację w dowolnym momencie. Mogę szybko przejść do kolejnych elementów, zachowując możliwość późniejszej zmiany.*

Dodatkowo abstrakcja kolekcji często kieruje moje zainteresowanie do tego, co „naprawdę" jest wykonywane, i odsuwa mnie od ścieżki implementowania określonego działania kolekcji, gdy tak naprawdę muszę mieć kilka bardzo prostych sposobów na wyszukanie tego, czego potrzebuję.

Ograniczone powtórzenia, wysoka ekspresywność i wczesne budowanie prostych abstrakcji. Tym jest dla mnie czysty kod.

W tych kilku akapitach Ron podsumował zawartość całej książki. Brak powtórzeń, jedna operacja, ekspresywność, małe abstrakcje. Mamy to wszystko.

Ward Cunningham, wynalazca Wiki, wynalazca Fit, współautor eXtreme Programming. Siła przewodnia dla wzorców projektowych. Lider Smalltalka i programowania obiektowego. Ojciec chrzestny wszystkich, którzy dbają o kod

Wiemy, że pracujemy na czystym kodzie, jeżeli każda procedura okazuje się taką, jakiej się spodziewaliśmy. Można nazywać go również pięknym kodem, jeżeli wygląda, jakby ten język został stworzony do rozwiązania danego problemu.

Tego typu zdania są charakterystyczne dla Warda. Czytasz je, kiwasz głową i przechodzisz do następnego tematu. Wygląda to tak rozsądnie, że prawie nie odbierasz tego jako czegoś dogłębnego. Możesz myśleć, że było to mniej więcej to, czego oczekiwałeś. Ale spójrzmy dokładniej.

„Jakiej się spodziewaliśmy" — kiedy ostatnio widziałeś moduł, który był w większości tym, czego się spodziewałeś? Czy nie trafiamy częściej na moduły, które wyglądają na zagmatwane, skomplikowane i niejasne? Nie jest to pogwałcenie tej zasady? 6Czy nie próbujesz często uchwycić i utrzymać wątki wnioskowania, które wypływają z całego systemu i torują sobie drogę przez czytany przez Ciebie moduł? Kiedy ostatnio czytałeś kod i kiwałeś głową z aprobatą tak, jak podczas czytania zasad Warda?

Ward oczekuje, że czytając czysty kod, nie będziemy zaskakiwani. W rzeczywistości nawet nie powinniśmy zbytnio poświęcać uwagi czytaniu. Po prostu powinniśmy czytać i będzie to mniej więcej to, czego oczekiwaliśmy. Powinno to być oczywiste, proste i zachęcające. Każdy moduł powinien przygotowywać grunt pod następny. Każdy powinien mówić nam, jak będzie napisany następny. Programy, które są tak czyste, są tak dobrze napisane, że niemal tego nie zauważamy. Projektant spowodował, że są niezwykle proste, jak wszystkie niezwykłe projekty.

Co możemy powiedzieć na temat uwagi Warda o pięknie? Wszyscy borykamy się z tym, że języki, którymi się posługujemy, nie zostały zaprojektowane specjalnie pod kątem naszych problemów projektowych. Jednak stwierdzenie Warda znów zrzuca na nas cały ciężar. Mówi on, że piękny kod *powoduje, że język wygląda tak, jakby był przeznaczony do rozwiązywania danego problemu*! Więc to na *nas* ciąży odpowiedzialność, by doprowadzić do tego, aby język był prosty! Uważajcie bigoci

językowi z wszystkich stron. To nie język powoduje, że program jest prosty. To programista sprawia, że język jest prosty!

Szkoły myślenia

Co ja (wujek Bob) mogę powiedzieć na ten temat? Czym według mnie jest czysty kod? Książka ta przedstawia, w najmniejszych szczegółach, co ja i moi przyjaciele myślimy na temat czystego kodu. Powiemy Ci, co uważamy za czystą nazwę zmiennej, czystą funkcję, czystą klasę i tak dalej. Przedstawimy te opinie jako bezwarunkowe i nie będziemy przepraszać za nasze ostre opinie. Dla nas, z punktu widzenia naszych karier, są one bezwarunkowe. Są one naszą *szkołą myślenia* o czystym kodzie.

Zwolennicy różnych sztuk walki nie zgadzają się w tym, która z nich jest najlepsza lub jaka jest najlepsza technika. Często mistrzowie sztuk walki tworzą swoje szkoły myślenia i zbierają uczniów, aby ich w dany sposób uczyć. Mamy więc *Gracie Jiu Jitsu*, założoną i prowadzoną przez rodzinę Gracie z Brazylii. Mamy *Hakkoryu Jiu Jitsu*, założoną i prowadzoną przez Okuyamę Ryuho z Tokio. Mamy *Jeet Kune Do*, założoną przez Bruce'a Lee w USA.

Uczniowie tych szkół zagłębiają się w naukach swoich mistrzów. Uczą się tego, co przedstawia określony mistrz, często wykluczając nauki innych mistrzów. Później, gdy zdobywają mistrzostwo w swojej sztuce, mogą stać się uczniami innych mistrzów, aby poszerzyć swoją wiedzę i praktykę. Niektórzy w końcu doskonalą swoje umiejętności, odkrywając nowe techniki, i zakładają nowe szkoły.

Żadna z tych różnych szkół nie ma *całkowitej racji*. Jednak wewnątrz określonej szkoły *działamy* tak, jakby nauki i techniki *były* prawidłowe. W końcu jest to prawidłowy sposób praktykowania technik prezentowanych w *Hakkoryu Jiu Jitsu* lub *Jeet Kune Do*. Jednak prawidłowość w danej szkole nie unieważnia nauki innych szkół.

Załóżmy, że książka ta ma podtytuł *Szkoła czystego kodu mentora obiektowego*. Znajdujące się w niej techniki i nauki pokazują, jak *my* praktykujemy *naszą* sztukę. Chcemy powiedzieć, że jeżeli będziesz stosował nasze techniki, będziesz cieszył się korzyściami, którymi my się cieszymy, i nauczysz się pisać kod czysty i profesjonalny. Jednak nie popełniaj błędu, myśląc, że jesteśmy nieomylni w bezwzględnym sensie. Istnieją inne szkoły i inni mistrzowie, którzy mają równie dużo cech profesjonalistów, co my. Uczenie się od nich również da Ci wiele dobrego.

W istocie wiele zaleceń zamieszczonych w tej książce jest kontrowersyjnych. Prawdopodobnie nie zgodzisz się z wszystkimi. Możesz je kwestionować. Świetnie. Nie uważamy się za ostateczny autorytet. Z drugiej strony, zalecenia zamieszczone w tej książce dotyczą technik, których poznanie zajęło nam dużo czasu. Nauczyliśmy się ich przez dekady doświadczenia oraz w wyniku wielokrotnych prób i błędów. Tak więc, niezależnie od tego, czy się zgadzasz, czy nie, byłoby błędem, gdybyś nie przyjął i nie docenił naszego punktu widzenia.

Jesteśmy autorami

Pole `@author` w Javadoc mówi nam, kim jesteśmy. Jesteśmy autorami. Autorzy mają zwykle czytelników. W rzeczywistości autorzy są *odpowiedzialni* za komunikację ze swoimi czytelnikami. Następnym razem, gdy napiszesz wiersz kodu, pamiętaj, że jesteś autorem piszącym dla czytelników, którzy będą oceniać Twoje wysiłki.

Można zadać pytanie, ile kodu naprawdę się czyta? Czy wysiłek w większości nie jest skupiony na jego pisaniu?

Czy kiedykolwiek odtwarzałeś sesję edycji? W latach osiemdziesiątych i dziewięćdziesiątych korzystaliśmy z takich edytorów jak Emacs, który zapamiętywał każde naciśnięcie klawisza. Można było popracować przez godzinę, a następnie odtworzyć całą sesję edycji jak w przyspieszonym filmie. Gdy to zrobiłem, efekty były fascynujące.

Ogromna większość procesu odtwarzania polegała na przewijaniu i przechodzeniu do innych modułów!

Bob wszedł do modułu.
Przewinął w dół, do funkcji wymagającej zmiany.
Przerwał, rozważając możliwości.
Przeszedł do początku modułu w celu sprawdzenia inicjalizacji zmiennej.
Teraz przesunął się w dół i zaczął pisać.
Ups, usunął to, co napisał!
Wpisał to jeszcze raz.
Usunął to jeszcze raz.
Wpisał połowę czegoś innego, ale usunął to!
Przesunął się do innej funkcji, która wywoływała zmienianą funkcję, aby sprawdzić, jak była wywoływana.
Przeszedł znów w górę i wpisał ten sam kod, który usunął.
Zatrzymał się.
Ponownie usunął ten kod.
Otworzył nowe okno i zajrzał do klasy pochodnej. Czy ta funkcja jest przesłonięta?

Wiesz już, o co chodzi. W rzeczywistości stosunek czasu spędzanego na czytaniu do czasu spędzanego na pisaniu jest jak 10:1. *Stale* czytamy starszy kod, jest to część pracy przy pisaniu nowego.

Ponieważ współczynnik jest tak wysoki, chcemy, aby czytanie kodu było łatwe, nawet jeżeli powoduje, że jego pisanie jest trudniejsze. Oczywiście, nie ma sposobu na pisanie kodu bez jego czytania, więc *ułatwienie czytania powoduje ułatwienie pisania*.

Nie ma od tego wyjątków. Nie można pisać kodu, jeżeli nie przeczyta się kodu go otaczającego. Kod, jaki próbujemy dziś napisać, może być trudny lub łatwy do napisania, w zależności od tego, jak łatwy lub trudny do czytania jest kod otaczający go. Tak więc jeżeli chcemy szybko iść naprzód, jeżeli chcemy szybko i łatwo pisać kod — musi być on łatwy w czytaniu.

Zasada skautów

Nie wystarczy dobrze pisać kod. Kod musi być *zachowany w czystości* przez cały czas. Wiele razy widzieliśmy kod, który z czasem psuł się i degradował. Dlatego należy aktywnie przeciwdziałać tej degradacji.

Skauci amerykańscy mieli prostą zasadę, jaką można zastosować w naszym zawodzie.

Pozostaw obóz czyściejszym, niż go zastałeś[5].

Jeżeli wszyscy będziemy dodawali do repozytorium kod nieco czyściejszy, niż ten, który pobraliśmy, po prostu nie będzie się psuł. Czyszczenie nie musi być duże. Wystarczy zmiana jednej nazwy zmiennej na lepszą, podział funkcji, która była nieco za duża, wyeliminowanie małego powtórzenia, wyczyszczenie jednej złożonej instrukcji `if`.

Czy można wyobrazić sobie pracę w projekcie, w którym kod po prostu *z czasem staje się lepszy*? Czy uważasz, że jakakolwiek inna opcja jest profesjonalna? Czy stała poprawa nie jest niezbędną częścią profesjonalizmu?

Poprzednik i zasady

W wielu miejscach książka ta jest „poprzedniczką" tej, którą napisałem w roku 2002, zatytułowanej *Agile Software Development: Principles, Patterns, and Practices* (PPP). Książka ta była poświęcona zasadom projektowania obiektowego i wielu praktykom stosowanym przez zawodowych programistów. Jeżeli nie czytałeś PPP, możesz odnieść wrażenie, że kontynuuje ona historię tej książki. Jeżeli już ją przeczytałeś, możesz zauważyć, że wiele jej fragmentów odbija się echem w tej książce, na poziomie kodu.

W tej książce znajdują się sporadyczne odwołania do różnych zasad projektowania. Mamy między innymi zasadę jednej odpowiedzialności (SRP), zasadę otwarty-zamknięty (OCP) oraz zasadę odwrócenia zależności (DIP). Zasady te są dokładnie opisane w PPP.

Zakończenie

Książki na temat sztuki nie obiecują, że staniesz się artystą. Wszystko, co mogą zrobić, to dać nam kilka narzędzi, technik i procesów myślenia wykorzystywanych przez artystów. Dlatego nie możemy obiecać, że każdy Czytelnik po przeczytaniu tej książki zostanie dobrym programistą. Nie możemy obiecać, że obdarzymy każdego intuicją potrzebną do tworzenia czystego kodu. Wszystko, co możemy zrobić, to przedstawić proces myślenia dobrego programisty oraz sztuczki, techniki i narzędzia przez niego wykorzystywane.

---

[5] Parafraza pożegnania skautów autorstwa Roberta Stephensona Smyth Baden-Powellsa: „Staraj się pozostawić ten świat nieco lepszym, niż go zastałeś".

Podobnie jak w przypadku książki na temat sztuki, również i ta jest wypełniona szczegółami. Jest w niej mnóstwo kodu. Pokażemy zarówno dobry, jak i zły kod. Pokażemy, jak można naprawić zły kod. Przedstawimy listę heurystyk, dyscyplin oraz technik. Pokażemy przykład po przykładzie. Później wszystko będzie w Twoich rękach.

Czy pamiętasz stary żart o skrzypku koncertowym? Zaczepił on starszego człowieka na skrzyżowaniu, pytając go, jak trafić do Carnegie Hall. Stary człowiek spojrzał na skrzypka i trzymane przez niego skrzypce i powiedział: „Trzeba ćwiczyć, synu. Ćwiczyć!".

Bibliografia

[Beck07]: Kent Beck, *Implementation Patterns*, Addison-Wesley 2007.

[Knuth92]: Donald E. Knuth, *Literate Programming*, Center for the Study of Language and Information, Leland Stanford Junior University 1992.

Znaczące nazwy

Tim Ottinger

Wstęp

Nazwy pełnią fundamentalną rolę w oprogramowaniu. Nazywamy nasze zmienne, funkcje, argumenty, klasy i pakiety. Nazywamy pliki źródłowe i katalogi, w których są umieszczone. Nazywamy pliki *jar*, *war* oraz *ear*. Nazywamy, nazywamy, nazywamy. Ponieważ robimy to tak często, powinniśmy robić to dobrze. W niniejszym rozdziale przedstawione są proste zasady tworzenia dobrych nazw.

Używaj nazw przedstawiających intencje

Łatwo powiedzieć, że nazwy powinny przedstawiać zamiary. Chcemy podkreślić, że traktujemy ten temat *naprawdę poważnie*. Wybór odpowiedniej nazwy to dobra inwestycja — zajmuje trochę czasu, ale pozwala oszczędzić go znacznie więcej. Dlatego warto przyglądać się używanym nazwom i zmieniać je, gdy znajdziemy lepsze. Każdy, kto czyta Twój kod (włącznie z Tobą), skorzysta na tym.

Nazwa zmiennej, funkcji lub klasy powinna być odpowiedzią na wszystkie ważne pytania. Powinna informować, w jakim celu istnieje, co robi i jak jest używana. Jeżeli nazwa wymaga komentarza, to znaczy, że nie ilustruje intencji.

```
int d; // Czas trwania w dniach
```

Nazwa `d` nic nam nie objaśnia. Nie przedstawia w żaden sensowny sposób czasu trwania ani dni. Powinniśmy wybrać nazwę, która przedstawia mierzoną wielkość oraz jednostkę pomiaru:

```
int elapsedTimeInDays;   // czasTrwaniawDniach
int daysSinceCreation;   // dniOdUtworzenia
int daysSinceModification;   // dniOdModyfikacji
int fileAgeInDays;   // wiekPlikuwDniach
```

Wybór nazw ilustrujących zamiary ułatwia zrozumienie i modyfikowanie kodu. Jakie jest przeznaczenie poniższego kodu?

```
public List<int[]> getThem() {
   List<int[]> list1 = new ArrayList<int[]>();
   for (int[] x : theList)
      if (x[0] == 4)
         list1.add(x);
   return list1;
}
```

Dlaczego tak trudno stwierdzić, co robi ten kod? Nie ma tu złożonych wyrażeń. Odstępy i wcięcia są rozsądne. Używane są w nim tylko trzy zmienne i dwie stałe. Nie ma tu nawet użytych żadnych fantazyjnych klas ani metod polimorficznych, jest tylko lista tablic.

Problemem nie jest prostota kodu, ale jego *niejawność* — kontekst kodu nie jest jasny na podstawie analizy samego kodu. W kodzie tym niejawnie zakładamy, że znamy odpowiedzi na takie pytania, jak:

1. Jakiego rodzaju elementy znajdują się w `theList`?
2. Jakie jest znaczenie zerowego indeksu elementu w `theList`?
3. Jakie jest znaczenie wartości `4`?
4. Do czego można użyć zwracanej listy?

W przykładowym kodzie nie ma odpowiedzi na te pytania, *ale mogą się one tam znajdować*. Załóżmy, że pracujemy nad grą w sapera. Odkryliśmy, że pole gry jest listą komórek o nazwie `theList`. Zmienimy więc jej nazwę na `gameBoard`.

Każda komórka pola gry jest reprezentowana przez prostą tablicę. Na podstawie dalszej analizy okazało się, że zerowy indeks lokacji zawiera wartość statusu, a wartość `4` oznacza, że komórka jest zaznaczona. Wystarczy nadać nazwy tym elementom i czytelność kodu znacznie się poprawi:

```
public List<int[]> getFlaggedCells() {
   List<int[]> flaggedCells = new ArrayList<int[]>();
   for (int[] cell : gameBoard)
   if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
   return flaggedCells;
}
```

Warto zwrócić uwagę, że prostota kodu została zachowana — nie zmieniła się ani liczba operatorów i stałych, ani liczba poziomów zagnieżdżenia. Mimo to kod stał się znacznie bardziej czytelny.

Możemy również pójść nieco dalej i napisać prostą klasę dla komórek w miejsce tablicy liczb `int`. Można utworzyć funkcję przedstawiającą intencje (nazwiemy ją `isFlagged`), pozwalającą ukryć magiczne liczby. Możemy więc napisać nową wersję funkcji:

```
public List<Cell> getFlaggedCells() {
   List<Cell> flaggedCells = new ArrayList<Cell>();
   for (Cell cell : gameBoard)
      if (cell.isFlagged())
         flaggedCells.add(cell);
   return flaggedCells;
}
```

Po tych prostych zmianach nazw nietrudno zrozumieć, o co chodzi w tym kodzie. W tym właśnie tkwi potęga wyboru odpowiednich nazw elementów kodu.

Unikanie dezinformacji

Programiści muszą wystrzegać się sugerowania fałszywych wskazówek zaburzających znaczenie kodu. Powinniśmy unikać słów, które wprowadzają znaczenia różniące się od oczekiwanego. Na przykład `hp`, `aix` czy `sco` będą nieodpowiednimi nazwami zmiennych, ponieważ są to nazwy platform Unix lub ich wariantów. Nawet jeżeli kodujemy obliczanie przeciwprostokątnej (ang. *hypotenuse*) i `hp` może wydawać się odpowiednim skrótem, może to być jednak dezinformujące.

Nie należy nazywać grupy kont mianem `accountList`, o ile nie jest to faktycznie zmienna typu `List`. Słowo „lista” ma dla programisty specjalne znaczenie. Jeżeli kontener przechowujący konta nie jest faktycznie typu `List`, może to prowadzić do fałszywych wniosków[1]. Dlatego lepiej użyć zmiennej `accountGroup`, `bunchOfAccounts` lub po prostu `accounts`.

Należy unikać nazw, które nieznacznie się od siebie różnią. Ile czasu zajmie zauważenie subtelnej różnicy pomiędzy zmienną `XYZControllerForEfficientHandlingOfStrings` w jednym module i znajdującą się nieco dalej `XYZControllerForEfficientStorageOfStrings`? Słowa te mają zbyt podobny wygląd.

---

[1] Jak się później okaże, nawet jeżeli kontener *jest* typu `List`, najlepiej nie wbudowywać typu kontenera w nazwę.

Przedstawianie podobnych koncepcji za pomocą podobnych sekwencji znaków to *informacja*. Używanie niespójnego nazewnictwa to *dezinformacja*. W nowoczesnych środowiskach Java można korzystać z automatycznego dokańczania kodu. Piszemy kilka znaków z nazwy i naciskamy kombinację klawiszy, co powoduje wyświetlenie listy możliwych zakończeń tej nazwy. Jest to bardzo pomocne, jeżeli nazwy bardzo podobnych elementów są posortowane alfabetycznie i jeżeli różnice są bardzo oczywiste, ponieważ programista może wybrać obiekt z użyciem nazwy bez konieczności przeglądania komentarzy, a nawet listy metod oferowanych przez klasę.

Przykładem niezwykle dezinformujących działań jest użycie w nazwach małej litery `l` oraz wielkiej `O`, szczególnie w połączeniach. Problem oczywiście polega na tym, że litery te wyglądają niemal tak jak cyfry jeden i zero.

```
int a = l;
if ( O == l )
   a = O1;
else
   l = O1;
```

Czytelnik może uważać, że jest to wymyślony przykład, ale osobiście wiele razy widzieliśmy kod, w którym takie zapisy są powszechne. W jednym przypadku autor kodu zasugerował użycie innej czcionki, dzięki której różnice będą bardziej widoczne, rozwiązanie to powinno być przekazane wszystkim nowym programistom jako tradycja przekazywana ustnie lub w postaci dokumentacji. Problem można rozwiązać ostatecznie bez dodatkowych nakładów — wystarczy zwykła zmiana nazwy.

Tworzenie wyraźnych różnic

Programiści często sami sobie tworzą problemy, gdy piszą kod pozwalający jedynie spełnić oczekiwania kompilatora lub interpretera. Ponieważ na przykład nie można użyć tej samej nazwy do tego samego elementu w zakresie, możemy ulec pokusie zmiany jednej nazwy w dowolny sposób. Czasami zdarza się to przez literówkę, co prowadzi do zaskakującej sytuacji, w której poprawienie błędu powoduje, że nie można skompilować kodu[2].

Nie wystarczy dodać numer seryjny lub dodatkowe słowo, nawet jeżeli wystarczy to kompilatorowi. Jeżeli nazwy muszą być różne, to powinny one również znaczyć coś innego.

Nazwy korzystające z numerów kolejnych (`a1`, `a2`, … `aN`) są przeciwieństwem nazw znaczących. Nazwy te nie są dezinformujące — są one nieinformujące; nie niosą ze sobą żadnej informacji o intencjach autora. Przeanalizujmy taki przykład:

[2] Weźmy jako przykład fatalną praktykę tworzenia zmiennej o nazwie `klass` tylko z tego powodu, że nazwa `class` jest zajęta.

```
public static void copyChars(char a1[], char a2[]) {
   for (int i = 0; i < a1.length; i++) {
      a2[i] = a1[i];
   }
}
```

Funkcja ta będzie znacznie czytelniejsza, jeżeli jako nazw argumentów użyjemy `source` i `destination`.

Dodatkowe słowa są kolejnym sposobem rozróżnienia nazw nieniosących żadnego znaczenia. Jako przykład weźmy klasę `Product`. Jeżeli utworzymy inną klasę, o nazwie `ProductInfo` lub `ProductData`, otrzymamy różne nazwy, ale nie spowodujemy, że będą one znaczyły coś innego. Słowa `Info` i `Data` są równie mało znaczącymi nazwami, jak `a`, `an` i `the`.

Oczywiście nie ma nic złego w korzystaniu z konwencji zalecających stosowanie przedrostków takich jak `a` czy `the`, pod warunkiem że zachowane są znaczące różnice. Na przykład możemy korzystać z przedrostka `a` dla wszystkich zmiennych lokalnych i `the` dla wszystkich argumentów funkcji[3]. Problem pojawia się w przypadku, gdy zdecydujemy się nazwać zmienną `theZork`, ponieważ istnieje już inna zmienna o nazwie `zork`.

Dodatkowe słowa są nadmiarowe. Słowo `variable` (lub `zmienna`, jeżeli przyjmiemy konwencję stosowania polskich nazw w kodzie) nigdy nie powinno pojawiać się w nazwie zmiennej. Słowo `table` (`tabela`) nigdy nie powinno pojawiać się w nazwie tabeli. W czym `NameString` jest lepsze od `Name`? Czy `Name` (`Nazwa`) będzie kiedykolwiek liczbą zmiennoprzecinkową? Jeżeli tak, złamana zostanie wcześniejsza zasada dotycząca dezinformacji. Wyobraźmy sobie, że znajdziemy jedną klasę o nazwie `Customer` i inną `CustomerObject`. Jak powinniśmy rozumieć różnicę pomiędzy nimi? Która reprezentuje najlepszą ścieżkę do historii płatności klienta?

Znamy aplikację, w której występują poniższe nazwy. Zostały one zmienione, aby chronić winnych, ale poniższe metody dokładnie ilustrują błąd:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Skąd programiści mają wiedzieć, którą z tych funkcji należy wywołać?

W przypadku braku specyficznych konwencji zmienna `moneyAmount` jest nieodróżnialna od `money`, `customerInfo` jest nieodróżnialna od `customer`, `accountData` jest nieodróżnialna od `account`, a `theMessage` nie da się odróżnić od `message`. Nazwy powinny różnić się w taki sposób, aby czytelnik wiedział, na czym polega różnica.

Tworzenie nazw, które można wymówić

Ludzie dobrze sobie radzą ze słowami. Znaczna część naszego mózgu jest stworzona do interpretacji słów. Słowa z definicji dają się wymówić. Marnotrawstwem byłoby nie skorzystać z dużej części naszych mózgów, które wyewoluowały w celu umożliwienia nam korzystania z języka mówionego. Dlatego powinniśmy tworzyć nazwy, które można wyartykułować.

---

[3] Wujek Bob korzystał z tej konwencji w C++, ale zarzucił tę praktykę, ponieważ nowoczesne IDE spowodowały, że jest niepotrzebna.

Jeżeli nie będziemy mogli ich wymówić, nie będzie można swobodnie dyskutować o kodzie. „Cóż, w bee cee arr three cee enn tee mamy pee ess zee kyew, widzisz?". Ma to znaczenie, ponieważ programowanie jest aktywnością społeczną.

Znam firmę korzystającą z nazw `genymdhms` (data generacji, rok, miesiąc, dzień, godzina, minuta i sekunda), więc często można było spotkać ludzi mamroczących „gen why emm dee aich emm ess". Ja mam zwyczaj — dla niektórych irytujący — wymawiania wszystkiego, co jest napisane, więc zacząłem czytać te nazwy „gen-yah-muddahims". Później zostało to podchwycone przez zespół projektantów i analityków, ale nadal brzmiało to głupio. Traktowaliśmy to jako dobry żart. Żart czy nie, tolerowaliśmy złe nazewnictwo. Konieczne było tłumaczenie nazw zmiennych nowym programistom, aż zaczęli korzystać z dziwnie brzmiących sztucznych słów, zamiast mówić zwykłymi angielskimi słowami. Porównajmy:

```
class DtaRcrd1O2 {
   private Date genymdhms;
   private Date modymdhms;
   private final String pszqint = "1O2";
   /* ... */
};
```

z:

```
class Customer {
   private Date generationTimestamp;
   private Date modificationTimestamp;;
   private final String recordId = "1O2";
   /* ... */
};
```

Możliwa stała się całkiem inteligentna konwersacja: „Marek spójrz na ten rekord! Znacznik czasu generacji jest ustawiony na jutrzejszą datę! Jak to się mogło stać?".

Korzystanie z nazw łatwych do wyszukania

Nazwy jednoliterowe i stałe numeryczne są szczególnie niewygodne, ponieważ nie można ich łatwo zlokalizować w tekście.

Bardzo łatwo można wyszukać ciąg znaków `MAX_CLASSES_PER_STUDENT`, ale liczba 7 może być problematyczna. Wyszukiwanie może zwrócić cyfry z nazw plików, innych definicji stałych lub wyrażeń, w których były użyte do innych celów. Jeszcze gorszym przypadkiem jest używanie długiej stałej numerycznej, w której ktoś przestawił cyfry, ponieważ błąd ten jest trudny do znalezienia przez programistę.

Podobnie nazwa `e` jest złym wyborem, jeżeli jest używana do zmiennej, którą programista będzie chciał kiedykolwiek znaleźć. Jest to najczęściej występująca litera w języku angielskim i najprawdopodobniej zostanie wyszukana w każdym fragmencie tekstu w każdym programie. Pod tym względem dłuższe nazwy przewyższają krótsze i każda dająca się wyszukać nazwa jest lepsza od stałej w kodzie.

Osobiście używam nazw jednoliterowych WYŁĄCZNIE jako zmiennych lokalnych wewnątrz krótkich metod. *Długość nazwy powinna odpowiadać rozmiarowi zasięgu* [N5]. Jeżeli zmienna lub stała może być widziana lub używana w wielu miejscach w kodzie, niezwykle ważne jest nadanie jej nazwy, którą można wyszukać. Kolejny raz porównajmy:

```
for (int j=0; j<34; j++) {
   s += (t[j]*4)/5;
}
```

z:

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
   int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
   int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
   sum += realTaskWeeks;
}
```

Warto zauważyć, że `sum` w powyższym przykładzie nie jest szczególnie użyteczną nazwą, ale przynajmniej daje się wyszukać. Kod z odpowiednimi nazwami jest dłuższy, ale zauważmy, że znacznie łatwiej znaleźć `WORK_DAYS_PER_WEEK` niż wszystkie miejsca, w których została użyta cyfra 5, i wśród nich te, które nas interesują.

Unikanie kodowania

Korzystamy na co dzień z wielu kodowań, więc nie ma potrzeby dodawania kolejnych. Kodowanie w nazwach typu lub informacji o zakresie powoduje zwiększenie nakładu pracy przy odszyfrowaniu. Trudno uznać za rozsądne wymaganie, aby każdy nowy pracownik musiał nauczyć się kolejnego „języka" kodowania oprócz zapoznania się z (zwykle rozsądną) treścią kodu, nad którym będzie pracował. Jest to niepotrzebne zakłócenie procesu myślenia w czasie rozwiązywania problemu. Kodowane nazwy zwykle trudno wymawiać, dlatego można się pomylić w ich wpisywaniu.

Notacja węgierska

W dawnych czasach, gdy korzystaliśmy z języków o długościach zmiennych stanowiących poważne wyzwania, z konieczności naruszaliśmy reguły. Fortran wymuszał kodowanie przez użycie pierwszej litery nazwy na kod typu. Wczesne wersje języka Basic pozwalały tylko na korzystanie z litery i jednej cyfry. Notacja węgierska (NW) pozwalała przejść na całkiem nowy poziom.

NW była uważana za bardzo ważną w API Windows C, gdy wszystko było uchwytem całkowitym, długim wskaźnikiem lub wskaźnikiem `void` albo jedną z kilku odmian „string" (o różnych zastosowaniach i atrybutach). W tym czasie kompilator nie sprawdzał typów, więc programiści potrzebowali sposobu na ich zapamiętanie.

W nowoczesnych językach mamy znacznie bogatszy system typów, a kompilator pamięta i wymusza typy. Co więcej, istnieje trend tworzenia mniejszych klas i krótszych funkcji, aby ludzie najczęściej widzieli punkt deklaracji każdej używanej zmiennej.

Programiści korzystający z języka Java nie potrzebują kodowania typów. Obiekty mają silnie wymuszane typy, a środowiska edycyjne są na tyle zaawansowane, że wykrywają błędy typów na długo przed uruchomieniem kompilatora! Tak więc NW i inne sposoby kodowania typów po prostu przeszkadzają. Powodują one, że trudniej zmienić nazwę lub typ zmiennej, funkcji lub klasy. Powodują one, że trudniej czyta się kod. Dodatkowo sprawiają, że system kodowania może mylić czytelnika.

```
PhoneNumber phoneString;
// Nazwa nie została zmieniona po zmianie typu!
```

Przedrostki składników

Nie potrzebujemy już przedrostków `m_` dla zmiennych składowych. Nasze klasy i funkcje powinny być na tyle małe, aby nie były one potrzebne. Powinniśmy używać środowiska edycyjnego, które wyróżnia lub koloruje składniki, dzięki czemu można je łatwo odróżnić.

```
public class Part {
   private String m_dsc; // Opis tekstowy
   void setName(String name) {
      m_dsc = name;
   }
}
```

```
public class Part {
   String description;
   void setDescription(String description) {
      this.description = description;
   }
}
```

Oprócz tego ludzie szybko uczą się ignorować przedrostki (lub przyrostki), aby widzieć znaczącą część nazwy. Im częściej czytamy kod, tym mniej widzimy przedrostków. W końcu staną się one niezauważalnym szumem i znacznikiem starszego kodu.

Interfejsy i implementacje

Czasami stosowane są specjalne przypadki kodowania. Załóżmy, że budujemy fabrykę abstrakcyjną (ang. *abstract factory*) do tworzenia figur. Fabryka ta będzie interfejsem implementowanym przez konkretną klasę. Jak powinniśmy ją nazwać? `IShapeFactory` i `ShapeFactory`? Osobiście pozostawiam nazwy interfejsów bez dekoracji. Początkowe `I`, tak częste w istniejącej bazie kodu, jest w najlepszym przypadku zakłóceniem, a w najgorszym nośnikiem zbyt dużej ilości informacji. Nie chcę, aby moi użytkownicy wiedzieli, że przekazuję im interfejs. Chcę, aby wiedzieli, że jest to `ShapeFactory`. Dlatego, jeżeli konieczne jest kodowanie interfejsu lub implementacji, wybieram kodowanie implementacji. Nazwanie tej klasy `ShapeFactoryImp`, a nawet w fatalny sposób `CShapeFactory`, jest lepsze niż kodowanie nazwy interfejsu.

Unikanie odwzorowania mentalnego

Czytelnicy nie powinni mentalnie przekształcać naszych nazw na inne, które znają. Problem ten wynika zwykle z powodu użycia terminów spoza domeny problemu lub domeny rozwiązania.

Jest to problem z jednoliterowymi nazwami zmiennych. Oczywiście, licznik pętli może mieć nazwę `i`, `j` lub `k` (ale nie `l`!), jeżeli zakres jest bardzo mały i nie ma nazw pozostających z nim w konflikcie. Można przyjąć to rozwiązanie, ponieważ jednoliterowe nazwy liczników pętli są już tradycją. Jednak w większości kontekstów nazwy jednoliterowe są złym wyborem; są to nazwy, które czytelnik musi mentalnie odwzorować na aktualną koncepcję. Nie ma gorszego powodu użycia nazwy `c` niż ten, że `a` i `b` są już zajęte.

Programiści są zwykle bystrymi ludźmi. Bystrzy ludzie czasami lubią pokazywać swoje możliwości przez demonstrowanie możliwości mentalnych. W końcu, jeżeli możemy niezawodnie zapamiętać, że `r` jest adresem URL zapisanym małymi literami z usuniętą nazwą hosta i schematu, to jasno widać, że jesteśmy bystrzy.

Jedną z różnic pomiędzy bystrymi programistami a profesjonalistami jest to, że profesjonaliści rozumieją, że *czytelność jest wszystkim*. Profesjonaliści piszą kod zrozumiały dla innych.

Nazwy klas

Klasy i obiekty powinny być rzeczownikami lub wyrażeniami rzeczownikowymi, takimi jak `Customer`, `WikiPage`, `Account` czy też `AddressParser`. Należy unikać w nazwach klas słów takich jak `Manager`, `Processor`, `Data` lub `Info`. Nazwy klas nie powinny być czasownikami.

Nazwy metod

Metody należy opatrywać nazwami będącymi czasownikami lub wyrażeniami czasownikowymi, takimi jak `postPayment`, `deletePage` lub `save`. Akcesory, mutatory i predykaty powinny mieć nazwy tworzone na podstawie nazwy obsługiwanej właściwości i być poprzedzone przedrostkiem `get`, `set` lub `is`, zgodnie ze standardem javabean[4].

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Gdy konstruktory są przeciążone, należy używać metod fabryk o nazwach opisujących argumenty. Na przykład:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

jest zwykle lepsze od

```
Complex fulcrumPoint = new Complex(23.0);
```

[4] *http://java.sun.com/products/javabeans/docs/spec.html*

Warto rozważyć wymuszenie stosowania tej techniki przez zadeklarowanie odpowiedniego konstruktora jako prywatnego.

Nie bądź dowcipny

Jeżeli nazwy są dowcipnymi szaradami, mogą być zapamiętane wyłącznie przez osoby z identycznym co autor poczuciem humoru, i tylko dopóty, dopóki będą one pamiętać dany żart słowny. Czy wszyscy będą wiedzieli, do czego służy funkcja o nazwie `HolyHandGrenade`? Oczywiście, jest to dowcipne, ale w tym przypadku `DeleteItems` będzie lepszą nazwą. Czytelność jest zawsze ważniejsza od wrażeń artystycznych.

Dowcipność w kodzie często przybiera formę kolokwializmów lub slangu. Na przykład nie należy używać nazwy `whack()`, jeżeli można `kill()`. Nie należy stosować żartów znanych w małych społecznościach, takich jak `eatMyShorts()` zamiast `abort()`.

Nazwa obiektu powinna odzwierciedlać jego znaczenie. Dowcipne hasła zachowajmy na spotkania z przyjaciółmi.

Wybieraj jedno słowo na pojęcie

Należy stosować zasadę *jedno słowo na jedno abstrakcyjne pojęcie* i trzymać się jej. Na przykład mylące jest stosowanie nazw `fetch`, `retrieve` i `get` do analogicznych metod w różnych klasach. Jak można zapamiętać, która metoda należy do której klasy? Niestety, często trzeba pamiętać, która firma, grupa lub osoba napisała bibliotekę lub klasę, aby zapamiętać, które nazwy zostały użyte. W przeciwnym razie spędzimy zbyt wiele czasu na przeglądaniu nagłówków i przykładów wcześniejszego kodu.

Nowoczesne środowiska edycyjne, takie jak Eclipse i IntelliJ, dostarczają podpowiedzi kontekstowych, takich jak lista metod, jakie można wywołać na danym obiekcie. Jednak warto pamiętać, że listy te nie zawierają zwykle komentarzy, jakie napisaliśmy obok nazw funkcji i list parametrów. Jeżeli będziemy mieli szczęście, będą tam *nazwy* parametrów z deklaracji funkcji. Nazwy funkcji muszą być niezależne i spójne, aby można było później wybrać właściwą metodę bez dodatkowych poszukiwań.

Podobnie mylące jest występowanie nazw `controller`, `manager` oraz `driver` w tej samej bazie kodu. Jaka jest najważniejsza różnica pomiędzy `DeviceManager` a `ProtocolController`? Dlaczego obie nazwy nie są menedżerami lub kontrolerami? Czy naprawdę obie są sterownikami? Nazwy te pozwalają oczekiwać, że dwa obiekty mają bardzo różne typy, jak również inne klasy.

Spójny leksykon jest ogromnym ułatwieniem dla programistów, którzy muszą korzystać z naszego kodu.

Nie twórz kalamburów!

Należy unikać używania tego samego słowa do dwóch celów. Używanie tego samego terminu dla dwóch różnych zagadnień jest właśnie kalamburem.

Jeżeli stosujemy zasadę „jedno słowo na zagadnienie", powinniśmy w efekcie otrzymać wiele klas, które zawierają na przykład metodę `add`. Jeżeli lista parametrów i zwracane wartości tych różnych metod `add` odpowiadają sobie składniowo, wszystko jest w najlepszym porządku.

Jednak ktoś może zdecydować, aby użyć „dla spójności" słowa `add`, gdy faktycznie wykonywana operacja nie jest dodawaniem. Załóżmy, że mamy wiele klas, w których `add` tworzy nową wartość przez dodawanie lub łączenie dwóch istniejących wartości. Załóżmy teraz, że piszemy nową klasę, która posiada metodę wstawiającą jeden parametr do kolekcji. Czy powinniśmy nazywać tę metodę `add`? Może się to wydawać spójne, ponieważ mamy tak dużo innych metod `add`, ale w tym przypadku działanie jest inne, więc powinniśmy użyć nazwy takiej jak `insert` lub `append`. Nazwanie tej nowej metody `add` byłoby kalamburem.

Naszym celem jako autorów powinno być uczynienie kodu tak łatwego do zrozumienia, jak jest to możliwe. Chcemy, aby nasz kod dało się szybko zrozumieć, a nie intensywnie studiować. Chcemy użyć popularnego modelu szkicowanego na kartce, w którym autor jest odpowiedzialny za czytelność przekazu, a nie akademickiego modelu, w którym zadaniem ucznia jest wydobycie wiedzy z papierowych tomów.

Korzystanie z nazw dziedziny rozwiązania

Należy pamiętać, że ludzie czytający nasz kod będą programistami. Można więc korzystać z terminów informatycznych, nazw algorytmów, nazw wzorców, terminów matematycznych i tak dalej. Niekoniecznie trzeba używać każdej nazwy z dziedziny problemu, ponieważ może to spowodować, że nasi współpracownicy będą musieli wielokrotnie pytać klienta, co oznacza dana nazwa, choć znają ten termin pod inną nazwą.

Nazwa `AccountVisitor` niesie ze sobą wiele informacji dla każdego programisty, który zna wzorzec *visitor*. Który programista nie będzie wiedział, co to jest `JobQueue`? Istnieje wiele technicznych zagadnień, które programiści muszą znać. Wybór nazwy technicznej dla tych elementów jest zwykle najlepszym rozwiązaniem.

Korzystanie z nazw dziedziny problemu

Wszędzie tam, gdzie nie istnieją terminy programistyczne dla wykonywanych operacji, należy używać nazw z dziedziny problemu. W najgorszym przypadku programista, który będzie konserwował kod, zapyta eksperta o znaczenie nazwy.

Rozdzielenie koncepcji domeny problemu i rozwiązania jest jednym z zadań dobrego programisty i projektanta. Kod, który jest bardziej zbliżony do koncepcji dziedziny problemu, powinien zawierać nazwy z tej dziedziny.

Dodanie znaczącego kontekstu

Istnieje niewiele nazw, które same w sobie niosą wystarczająco dużo treści — w większości przypadków tak nie jest. Z tego powodu konieczne jest umieszczenie nazw we właściwym kontekście przez wstawienie ich w odpowiednio nazwane klasy, funkcje i przestrzenie nazw. Gdy wszystko inne zawiedzie, ostatnią deską ratunku może być zastosowanie przedrostka nazwy.

Wyobraźmy sobie, że mamy zmienne o nazwach `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` oraz `zipcode`. Jeżeli są one zgrupowane, jasne jest, że tworzą adres. Co w przypadku, gdy w metodzie zobaczymy pojedynczą zmienną `state`? Czy automatycznie skojarzymy ją z częścią adresu?

Można utworzyć kontekst przez zastosowanie przedrostka: `addrFirstName`, `addrLastName`, `addrState` i tak dalej. Dzięki temu czytelnicy będą wiedzieli, że zmienne te są częścią większej struktury. Oczywiście, lepszym rozwiązaniem jest utworzenie klasy o nazwie `Address`. W takim przypadku nawet kompilator będzie wiedział, że zmienna jest częścią większej całości.

Przeanalizujmy metodę z listingu 2.1. Czy zmienne wymagają lepszego kontekstu?

Nazwa funkcji tworzy tylko część kontekstu; algorytm zapewnia pozostałą część. Gdy zapoznamy się z funkcją, okaże się, że trzy zmienne, `number`, `verb` oraz `pluralModifier`, są częścią komunikatu „obliczanej statystyki”. Niestety, konieczne jest wywnioskowanie kontekstu. Gdy po raz pierwszy spojrzymy na funkcję, znaczenie tych zmiennych nie będzie jasne.

LISTING 2.1. Zmienne o niejasnym kontekście

```
private void printGuessStatistics(char candidate, int count) {
   String number;
   String verb;
   String pluralModifier;
   if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
   } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
   } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
   }
   String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier);
   print(guessMessage);
}
```

Funkcja jest zbyt duża, a zmienne są intensywnie używane. Aby podzielić funkcję na mniejsze elementy, musimy utworzyć klasę `GuessStatisticsMessage`, której składowymi będą te trzy zmienne. Zapewnia to jasny kontekst dla zmiennych. Są one *jednoznacznie* częścią klasy `Guess ↪StatisticsMessage`. Poprawienie kontekstu powoduje również, że algorytm jest znacznie bardziej czytelny, ponieważ jest podzielony na mniejsze funkcje (patrz listing 2.2).

LISTING 2.2. Zmienne z kontekstem

```
public class GuessStatisticsMessage {
   private String number;
   private String verb;
   private String pluralModifier;

   public String make(char candidate, int count) {
      createPluralDependentMessageParts(count);
      return String.format(
         "There %s %s %s%s",
         verb, number, candidate, pluralModifier );
   }

   private void createPluralDependentMessageParts(int count) {
      if (count == 0) {
         thereAreNoLetters();
      } else if (count == 1) {
         thereIsOneLetter();
      } else {
         thereAreManyLetters(count);
      }
   }

   private void thereAreManyLetters(int count) {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
   }

   private void thereIsOneLetter() {
      number = "1";
      verb = "is";
      pluralModifier = "";
   }

   private void thereAreNoLetters() {
      number = "no";
      verb = "are";
      pluralModifier = "s";
   }
}
```

Nie należy dodawać nadmiarowego kontekstu

W hipotetycznej aplikacji o nazwie „Luksusowa stacja benzynowa" nie należy prefiksować każdej klasy skrótem `LSB`. Po prostu będziemy mieli przeciwko sobie używane narzędzia. Wpisujemy `L`, naciskamy klawisz dokańczania i otrzymujemy długą na kilometr listę wszystkich klas w systemie. Czy jest to mądre? Dlaczego utrudniamy sobie pracę z IDE?

Załóżmy, że w module księgowym `LSB` utworzyliśmy klasę `MailingAddress` i nazwaliśmy ją `LSBAccountAddress`. Później potrzebujemy adresu wysyłki dla aplikacji obsługującej kontakty z klientami. Czy użyjemy klasy `LSBAccountAddress`? Czy będzie to prawidłowa nazwa? Dziesięć z siedemnastu znaków to znaki nadmiarowe lub nieodpowiednie.

Krótsze nazwy są zwykle lepsze niż dłuższe, o ile są one jasne. Nie należy opatrywać nazwy nadmiarowym kontekstem.

Nazwy `accountAddress` i `customerAddress` są doskonałe dla instancji klasy `Address`, ale mogą być kiepskimi nazwami dla klas. `Address` jest świetną nazwą dla klasy. Jeżeli muszę odróżniać adresy MAC, adresy portów i adresy WWW, mogę używać nazw `PostalAddress`, `MAC` oraz `URI`. Wynikowe nazwy są bardziej precyzyjne, co jest ważne.

Słowo końcowe

Wybór dobrych nazw wymaga umiejętności opisywania i podstaw kulturowych. Trzeba się tego nauczyć, ale nie jest to problemem techniczny, biznesowy ani organizacyjny. Niestety, wiele osób z branży nie posiada tych umiejętności.

Ludzie często nie zmieniają nazw elementów z obawy, że inni programiści będą mieli zastrzeżenia. My nie mamy takich obaw i uważamy, że zmiany nazw na czytelniejsze to zawsze dobry pomysł. W większości przypadków nie pamiętamy nazw klas i metod. Korzystamy z nowoczesnych narzędzi, które obsługują tego typu szczegóły, dzięki czemu możemy skupić się na tym, by kod można było czytać jak akapity i zdania, a co najmniej jak tabele i struktury danych (zdania nie zawsze nadają się do wyświetlania danych). Prawdopodobnie zaskoczymy kogoś, gdy zmienimy nazwę, podobnie jak w przypadku innych usprawnień kodu, ale nie powstrzymuje to nas przed dokonywaniem zmian.

Skorzystaj z przedstawionych tu zasad i sprawdź, jak wpływają one na czytelność tworzonego kodu. Jeżeli konserwujesz kod napisany przez kogoś innego, możesz skorzystać z narzędzi *refactoringu*, które pomagają rozwiązywać problemy związane ze zmianą nazwy.

Funkcje

W ZAMIERZCHŁYCH CZASACH PROGRAMOWANIA tworzyliśmy systemy z programów i podprogramów. Następnie, w erze Fortrana i PL/1, korzystaliśmy z programów, podprogramów i funkcji. Tylko funkcje przeszły próbę czasu. Funkcje są pierwszą linią organizacji każdego programu. Tematem tego rozdziału jest właśnie pisanie dobrych funkcji.

Przeanalizujmy kod z listingu 3.1. Trudno jest znaleźć długą funkcję w FitNesse[1], ale gdy zagłębiłem się w kod, znalazłem taką. Nie tylko jest długa, ale także zawiera powielony kod, sporo dziwnych tekstów, wiele obcych i nieoczywistych typów danych oraz API. Sprawdźmy, o ile bardziej sensowna może być po poświęceniu jej trzech minut.

LISTING 3.1. HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
```

[1] Narzędzie testujące dostępne na zasadach open source, *www.fitnese.org*.

```
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Czy można zrozumieć działanie tej funkcji po trzech minutach analizowania? Prawdopodobnie nie. Zbyt wiele tu się dzieje na zbyt wielu poziomach abstrakcji. Mamy tu nieznane napisy i dziwne wywołania funkcji wymieszane z podwójnie zagnieżdżonymi instrukcjami `if` kontrolowanymi przez znaczniki.

Jednak po wyodrębnieniu kilku metod, zmianie kilku nazw i niewielkiej modyfikacji struktury byłem w stanie zawrzeć przeznaczenie tej funkcji w dziewięciu wierszach przedstawionych na listingu 3.2. Czy *to* da się zrozumieć w trzy minuty?

LISTING 3.2. HtmlUtil.java (zmodyfikowany)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
    return pageData.getHtml();
}
```

O ile nie spędziliśmy więcej czasu na studiowaniu FitNesse, prawdopodobnie nie zrozumiemy wszystkich szczegółów. Jednak można wywnioskować, że ta funkcja dołącza pewne strony konfiguracji i rozbioru do testowanej strony, a następnie generuje tę stronę w HTML-u. Czytelnik znający JUnit[2] zapewne zorientuje się, że funkcja ta należy do pewnego rodzaju biblioteki testującej bazującej na WWW. Wydzielenie tej informacji z listingu 3.2 jest dosyć łatwe, ale na listingu 3.1 jest ona znacznie bardziej ukryta.

Co powoduje, że funkcje takie jak na listingu 3.2 są czytelne i zrozumiałe? Co można zrobić, by funkcja komunikowała swoje przeznaczenie? Jakie atrybuty możemy nadać funkcjom, aby przypadkowy czytelnik mógł wywnioskować, w jakiego rodzaju programie się znajdują?

---

[2] Narzędzie do testowania jednostkowego dostępne na zasadach open source dla języka Java, *www.junit.org*.

Małe funkcje!

Pierwsza zasada dotycząca konstruowania funkcji jest taka, że powinny być małe. Druga zasada mówi, że *powinny być mniejsze, niż są*. Nie jest to założenie, które mogę udowodnić. Nie mogę przedstawić przykładów badań pokazujących, że bardzo małe funkcje są lepsze. Mogę jedynie powiedzieć, że przez niemal czterdzieści lat napisałem funkcje o bardzo różnych rozmiarach. Napisałem kilka paskudnych 3000-wierszowych potworów. Napisałem również wiele funkcji mających od 100 do 300 wierszy. Pisałem też funkcje mające od 20 do 30 wierszy. Po wielu próbach i błędach mogę powiedzieć, że funkcje powinny być bardzo małe.

W latach osiemdziesiątych ubiegłego wieku panował pogląd głoszący, że funkcja powinna mieścić się na jednym ekranie. Oczywiście, mówiliśmy tak, gdy terminal VT100 miał 24 wiersze po 80 kolumn, a nasze edytory zabierały cztery wiersze na cele administracyjne. Obecnie, przy zastosowaniu niewielkiej czcionki i dużego monitora, można zmieścić 150 znaków w wierszu i co najmniej 100 wierszy na ekranie. Wiersze jednak nie powinny mieć po 150 znaków. Funkcje nie powinny mieć 100 wierszy długości. Funkcje powinny mieć właśnie nie więcej niż 20 wierszy.

Jak krótkie powinny być funkcje? W 1999 roku odwiedziłem Kenta Becka w jego domu w Oregonie. Kent pokazał mi wówczas mały program Java/Swing, który nazwał *Sparkle*. Tworzył on efekty graficzne bardzo podobne do tych, jakie widzieliśmy przy różdżce wróżki w filmie *Kopciuszek*. Gdy przesuwałem wskaźnik myszy, ciągnął on za sobą iskrzący się ślad, a iskry te, poddawane symulowanemu polu grawitacyjnemu, opadały na dół okna. Gdy Kent pokazał mi kod realizujący ten efekt, byłem zaszokowany — wszystkie funkcje były niezwykle krótkie. Byłem przyzwyczajony do tego, że funkcje w programach Swing miały całe kilometry długości. Każda funkcja w *tym* programie miała dwa, trzy lub cztery wiersze długości. Każda była przejrzysta i oczywista. Każda miała do opowiedzenia jakąś historię. Każda prowadziła do następnej we wzorowym porządku. *Tak krótkie* powinny być funkcje w naszych programach[3]!

A zatem, jakie rozmiary powinny mieć tworzone przez nas funkcje? Powinny być zwykle krótsze niż ta z listingu 3.2! Kod z listingu 3.2 powinien być skrócony do postaci z listingu 3.3.

LISTING 3.3. HtmlUtil.java (ponownie zmodyfikowany)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

[3] Pytałem Kenta, czy ma jeszcze ten plik, ale nie mógł go znaleźć. Przeszukiwałem również moje stare komputery, ale nie znalazłem go. Program pozostał więc tylko w mojej pamięci.

Bloki i wcięcia

Z powyższych założeń wynika, że bloki w instrukcjach `if`, `else`, `while` i podobnych powinny mieć po jednym wierszu. Prawdopodobnie wiersze te powinny być wywołaniem funkcji. Nie tylko pozwala to zachować niewielką objętość całej funkcji, ale również ma wartość dokumentacyjną, ponieważ funkcja wywołana w bloku może mieć sugestywną, opisową nazwę.

Oznacza to, że funkcja nie może być na tyle duża, aby zawierała zagnieżdżone struktury. Dlatego poziom wcięć w funkcji nie powinien przekraczać dwóch. Dzięki temu funkcje są czytelniejsze i bardziej zrozumiałe.

Wykonuj jedną czynność

Kod z listingu 3.1 wykonuje kilka czynności: tworzy bufory, pobiera strony, wyszukuje strony dziedziczone, generuje ścieżki, dołącza tajemnicze napisy i generuje kod HTML. Ogrom różnych czynności tworzy wielki bałagan. Z drugiej strony, kod z listingu 3.3 wykonuje jedną prostą operację. Dołącza strony konfiguracyjne i rozbioru do testowanych stron.

Poniższa porada pojawia się w takiej lub innej formie już co najmniej od 30 lat.

> ***FUNKCJE POWINNY WYKONYWAĆ JEDNĄ OPERACJĘ. POWINNY ROBIĆ TO DOBRZE. POWINNY ROBIĆ TYLKO TO.***

W przypadku tej porady problem polega na tym, że nie wiadomo, co to jest „jedna operacja". Czy kod z listingu 3.3 wykonuje jedną operację? Łatwo wykazać, że wykonywane są trzy czynności:

1. Sprawdzenie, czy strona jest stroną testową.
2. Jeżeli tak, dołączenie stron konfiguracyjnych i rozbioru.
3. Wygenerowanie strony w HTML-u.

Jak to w końcu jest? Czy ta funkcja wykonuje jedną operację, czy trzy? Należy zwrócić uwagę, że te trzy kroki z funkcji znajdują się o jeden poziom abstrakcji poniżej zadeklarowanego przez nazwę funkcji. Możemy opisać naszą funkcję za pomocą krótkiego akapitu *OTO*[4]:

> *OTO RenderPageWithSetupsAndTeardowns, sprawdzamy w niej, czy strona jest stroną testową, i jeżeli tak, dołączamy konfigurację i rozbiór. W każdym z przypadków generujemy stronę w HTML-u.*

[4] W języku programowania LOGO używano słowa kluczowego `OTO` do tych samych celów, do jakich języki Ruby i Python korzystały ze słowa `def`. Każda funkcja zaczynała się od słowa `OTO`. Ma to interesujący wpływ na sposób projektowania funkcji.

Jeżeli funkcja wykonuje tylko operacje znajdujące się o jeden poziom poniżej zadeklarowanej nazwy funkcji, to wykonuje ona jedną operację. W końcu powodem pisania funkcji jest dekompozycja dużych zagadnień (inaczej mówiąc, nazwy funkcji) na zbiór operacji znajdujących się na niższych poziomach abstrakcji.

Z listingu 3.1 powinno jasno wynikać, że zawiera on operacje z wielu różnych poziomów abstrakcji. Wynika z tego, że funkcja wykonuje więcej niż jedną operację. Nawet na listingu 3.2 znajdują się dwa poziomy abstrakcji, co pozwoliło nam na jego zmniejszenie. Jednak bardzo trudno jest sensownie skrócić listing 3.3. Możemy wyodrębnić instrukcję `if` do funkcji o nazwie `includeSetupsAndTeardownsIfTestPage`, ale to zmienia tylko strukturę kodu, a nie liczbę poziomów abstrakcji.

Tak więc innym sposobem sprawdzenia, czy funkcja wykonuje tylko „jedną operację", jest próba wyodrębnienia z niej innej funkcji, która nie jest tylko reorganizacją jej implementacji [G34].

Sekcje wewnątrz funkcji

Spójrzmy na listing 4.7 z następnego rozdziału. Należy zwrócić uwagę, że funkcja `generatePrimes` jest podzielona na sekcje, takie jak *deklaracje*, *inicjalizacja* oraz *sito*. Jest to oczywisty symptom wykonywania więcej niż jednej operacji. Funkcji wykonującej jedną operację nie można w rozsądny sposób podzielić na sekcje.

Jeden poziom abstrakcji w funkcji

Aby upewnić się, że nasze funkcje wykonują „jedną operację", musimy sprawdzić, czy instrukcje w funkcji są na tym samym poziomie abstrakcji. Łatwo zauważyć, że na listingu 3.1 reguła ta nie jest zachowana. Są tu koncepcje znajdujące się na wysokim poziomie abstrakcji, takie jak `getHTML()`; inne na średnim poziomie abstrakcji, takie jak `String pagePathName = PathParser.render(pagePath)`; jeszcze inne na wyraźnie niskim poziomie, takie jak `append("\n")`.

Mieszanie poziomów abstrakcji w jednej funkcji zawsze jest mylące. Czytelnicy mogą mieć problemy z rozpoznaniem, czy określone wyrażenie jest ważnym zagadnieniem, czy mało istotnym szczegółem. Co gorsza, gdy szczegóły wymieszają się z ważnymi koncepcjami, rodzi się pokusa dodawania do funkcji coraz większej liczby szczegółów.

Czytanie kodu od góry do dołu — zasada zstępująca

Chcemy, aby kod można było czytać od góry do dołu[5]. Chcemy, aby po każdej funkcji znajdowała się kolejna, na następnym poziomie abstrakcji, dzięki czemu można czytać program, schodząc o jeden poziom abstrakcji niżej wraz z przejściem do kolejnej funkcji w pliku. Nazywam to *zasadą zstępującą*.

---

[5] [KP78].

Inaczej mówiąc, chcemy, aby można było czytać program tak, jakby był zbiorem akapitów *ABY*, z których każdy opisuje bieżący poziom abstrakcji odwołujących się do kolejnych akapitów *ABY* na następnym poziomie.

Aby dołączyć konfiguracje i rozbiory, dołączamy konfigurację, następnie zawartość strony testowej, po czym rozbiory.

Aby dołączyć konfigurację, dołączamy konfigurację zestawu, jeżeli jest to zestaw, a następnie zwykłą konfigurację.

Aby dołączyć konfigurację zestawu, wyszukujemy w hierarchii nadrzędnej stronę „SuiteSetUp" i dodajemy instrukcję `include` ze ścieżką do tej strony.

Aby przeszukać hierarchię nadrzędną…

Praktyka pokazuje, że programiści mają problemy z przyswojeniem sobie tej reguły i pisaniem funkcji działających na jednym poziomie abstrakcji. Programista, który chce być profesjonalistą, powinien jednak wypracować sobie nawyk tworzenia krótkich funkcji, które na pewno wykonują „jedną operację". Konstruowanie kodu w taki sposób, aby można było czytać go od góry do dołu jak serię akapitów *ABY*, jest efektywną techniką zachowywania spójności poziomu abstrakcji.

Zwróćmy uwagę na listing 3.7, znajdujący się na końcu tego rozdziału. Zawiera on całą funkcję `testableHtml` przebudowaną zgodnie z opisanymi tu zasadami. Warto zwrócić uwagę, że każda funkcja wprowadza następny poziom abstrakcji i każda pozostaje w obrębie jednego poziomu.

Instrukcje switch

Trudno jest napisać małą instrukcję `switch`[6]. Nawet jeżeli zawiera ona tylko dwa przypadki, jest większa niż oczekiwana przeze mnie wielkość jednego bloku kodu lub funkcji. Równie trudno jest napisać instrukcję `switch` wykonującą jedną operację. Ze swojej natury instrukcje `switch` zawsze wykonują *n* operacji. Niestety, nie zawsze można uniknąć stosowania instrukcji `switch`, ale możemy zapewnić, że każda z nich jest umieszczona w niskopoziomowej klasie i nigdy nie jest powtarzana. Możemy to zrobić przy wykorzystaniu polimorfizmu.

Rozważmy kod z listingu 3.4. Mamy tu tylko jedną operację, która może zależeć od typu pracownika.

LISTING 3.4. Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
```

[6] Oczywiście dotyczy to również łańcuchów `if-else`.

```
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

Z funkcją tą wiąże się kilka problemów. Po pierwsze, jest duża, a po dodaniu kolejnego typu pracownika urośnie jeszcze bardziej. Po drugie, jasne jest, że wykonuje więcej niż jedną operację. Po trzecie, narusza zasadę pojedynczej odpowiedzialności[7] (SRP), ponieważ istnieje więcej niż jeden powód uzasadniający zmianę. Po czwarte, narusza zasadę otwarty-zamknięty[8] (OCP), ponieważ musi się zmieniać przy każdym dodaniu nowego typu. Jednak najprawdopodobniej największym problemem w przypadku tej funkcji jest to, że istnieje duża liczba innych funkcji mających taką samą strukturę. Możemy mieć na przykład:

```
isPayday(Employee e, Date date),
```

lub

```
deliverPay(Employee e, Money pay),
```

i wiele innych. Wszystkie te funkcje będą miały taką samą niewłaściwą strukturę.

Rozwiązaniem tego problemu jest ukrycie instrukcji `switch` w podstawie *fabryki abstrakcyjnej*[9] (patrz listing 3.5), co spowoduje, że nigdy już jej nie zobaczymy. Fabryka użyje instrukcji `switch` do tworzenia odpowiednich obiektów typów bazujących na `Employee`, a różne funkcje, takie jak `calculatePay`, `isPayday` czy `deliverPay`, będą rozmieszczone polimorficznie przez interfejs `Employee`.

Moją naczelną zasadą dotyczącą stosowania instrukcji `switch` jest tolerowanie wyłącznie jednego wystąpienia podczas tworzenia obiektów polimorficznych i ich ukrywanie w relacji dziedziczenia, aby pozostałe części systemu ich nie widziały [G23]. Oczywiście każdy przypadek jest inny i zdarza się, że naruszam tę zasadę w jednym lub kilku miejscach.

LISTING 3.5. Pracownicy i fabryka

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
-----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
```

---

[7] a) *http://en.wikipedia.org/wiki/Single_responsibility_principle*
b) *http://www.objectmentor.com/resources/articles/srp.pdf*

[8] a) *http://en.wikipedia.org/wiki/Open/closed_principle*
b) *http://www.objectmentor.com/resources/articles/ocp.pdf*

[9] [GOF].

```
      switch (r.type) {
        case COMMISSIONED:
          return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
      }
    }
  }
```

Korzystanie z nazw opisowych

Na listingu 3.7 zmieniliśmy nazwę naszej przykładowej funkcji z `testableHtml` na `SetupTear`
`↪downIncluder.render`. Jest to lepsza nazwa, ponieważ dokładniej opisuje przeznaczenie funkcji. Każdej z metod prywatnych nadałem równie opisową nazwę, taką jak `isTestable` czy `include`
`↪SetupAndTeardownPages`. Trudno nie doceniać wartości dobrych nazw. Warto zapamiętać zasadę Warda: *Wiemy, że pracujemy na czystym kodzie, jeżeli każda procedura okazuje się taką, jakiej się spodziewaliśmy*. Połową sukcesu w osiągnięciu tego stanu jest wybór dobrych nazw dla małych funkcji wykonujących jedną operację. Im mniejsze i lepiej ukierunkowane są funkcje, tym łatwiej wybrać dla nich opisową nazwę.

Nie należy obawiać się konstruowania długich nazw. Długa opisowa nazwa jest lepsza niż krótka enigmatyczna. Długa opisowa nazwa jest lepsza niż długi opisowy komentarz. Należy użyć konwencji nazewnictwa, która umożliwia łatwe odczytywanie wielu słów w nazwie funkcji, a następnie wykorzystanie tych wielu słów do nadania funkcji nazwy, która informuje o przeznaczeniu funkcji.

Czas poświęcony na wybór odpowiedniej nazwy jest dobrą inwestycją. Warto wypróbować różne nazwy i przeczytać kod korzystający z każdej z nich. Nowoczesne środowiska IDE, takie jak Eclipse lub IntelliJ, znacznie ułatwiają zmianę nazwy. Warto użyć tych mechanizmów i wypróbować różne nazwy.

Wybór opisowych nazw pozwala na wyjaśnienie projektu modułu i pomaga w jego usprawnianiu. Często zdarza się, że polowanie na dobrą nazwę skutkuje zmianą struktury kodu na lepszą.

Nazwy powinny być spójne. W funkcjach należy używać tych samych fraz, rzeczowników i czasowników, jakie wybraliśmy dla modułów. Jako przykład weźmy nazwy `includeSetupAndTear`
`↪downPages`, `includeSetupPages`, `includeSuiteSetupPage` i `includeSetupPage`. Podobna frazeologia w tych samych nazwach pozwala na utworzenie sekwencji opowiadającej historię. Faktycznie, jeżeli zobaczymy powyższą sekwencję, zapytamy się: „Co się dzieje w `includeTeardownPages`, `includeSuiteTeardownPage` oraz `includeTeardownPage`?”. Odpowiedź powinna brzmieć: „W większości to, czego się spodziewamy”.

Argumenty funkcji

Idealną liczbą argumentów dla funkcji jest zero (funkcja bezargumentowa). Następnie mamy jeden (jednoargumentowa) i dwa (dwuargumentowa). Należy unikać konstruowania funkcji o trzech argumentach (trzyargumentowych). Więcej niż trzy argumenty (funkcja wieloargumentowa) wymagają specjalnego uzasadnienia — a nawet wtedy takie funkcje nie powinny być stosowane.

Argumenty są kłopotliwe. Wymagają one użycia sporej ilości energii koncepcyjnej. Z tego powodu usunę niemal wszystkie argumenty z naszego przykładu. Weźmy jako przykład `String ↪Buffer`. Możemy przekazywać ten obiekt jako argument zamiast przechowywać go jako zmienną instancyjną, ale nasi czytelnicy będą musieli go interpretować za każdym razem, gdy go zobaczą. Gdy czytamy historię, jaką opowiada moduł, `includeSetupPage()` jest łatwiejsze do zrozumienia niż `includeSetup ↪PageInto(newPageContent)`. Argumenty znajdują się na innym poziomie abstrakcji niż funkcje i wymusza to zapoznanie się ze szczegółami (inaczej mówiąc, ze `StringBuffer`), co nie jest w tym miejscu szczególnie istotne.

Argumenty są jeszcze bardziej kłopotliwe z punktu widzenia testowania. Trudno napisać wszystkie testy jednostkowe zapewniające, że wszystkie kombinacje argumentów będą działały prawidłowo. Jeżeli nie ma argumentów, jest to bardzo proste. Jeżeli mamy jeden argument, nie jest to zbyt trudne. Przy dwóch argumentach problem staje się bardziej kłopotliwy. Jeżeli mamy więcej niż dwa argumenty, testowanie każdej kombinacji odpowiednich wartości może być nużące.

Argumenty wyjściowe są trudniejsze do zrozumienia niż argumenty wejściowe. Gdy czytamy funkcje, zwykle zakładamy, że dane *wchodzą* do funkcji przez argumenty i *wychodzą* z funkcji poprzez zwracaną wartość. Zwykle nie oczekujemy, że dane będą wychodziły z funkcji przez argumenty. Z tego powodu argumenty wyjściowe powodują często konieczność powtarzania analizy.

Jeden argument jest prawie tak dobry jak brak argumentów. Funkcja `SetupTeardownIncluder. ↪render(pageData)` jest bardzo łatwa do zrozumienia. Jasne jest, że mamy zamiar *wygenerować* dane z obiektu `pageData`.

Często stosowane funkcje jednoargumentowe

Istnieją dwa często spotykane powody przekazania jednego argumentu do funkcji. Możemy zadawać pytanie na temat tego argumentu, tak jak w `boolean fileExists("MyFile")`. Można również operować na argumencie, przekształcając go w coś innego i *zwracając wynik*. Na przykład `InputStream fileOpen("NazwaPliku")` powoduje przekształcenie nazwy pliku typu `String` na zwracany obiekt `InputStream`. Są to dwa zastosowania, jakich czytelnik oczekuje od funkcji. Należy używać nazw, które jasno definiują działanie, i zawsze używać tych postaci w spójny sposób (patrz „Rozdzielanie poleceń i zapytań” w dalszej części rozdziału).

Nieco rzadziej spotykaną, ale również przydatną odmianą funkcji jednoargumentowej jest *zdarzenie*. W tej postaci funkcja posiada argument wejściowy, ale nie ma wyjściowego. W programie interpretuje się takie funkcje jako wywołania zdarzeń i wykorzystuje argumenty do zmiany stanu systemu, na przykład `void passwordAttemptFailedNtimes(int attempts)`. Nie należy nadużywać funkcji w tej postaci. Czytelnik musi być jasno poinformowany, że jest to zdarzenie. Trzeba uważnie wybierać nazwę i kontekst.

Należy unikać funkcji jednoargumentowych, które nie należą do żadnej z tych postaci, na przykład `void includeSetupPageInto(StringBuffer pageText)`. Użycie argumentu wyjściowego zamiast zwracanej wartości dla transformacji jest mylące. Jeżeli funkcja ma przekształcać argument wejściowy, wynik tej transformacji powinien być wartością zwracaną. Faktycznie, `StringBuffer transform(StringBuffer in)` jest lepsza niż `void transform(StringBuffer out)`, nawet jeżeli w implementacji pierwszej funkcji po prostu zwracamy argument wejściowy. Przynajmniej ma to postać transformacji.

Argumenty znacznikowe

Argumenty znacznikowe są brzydkie. Przekazanie wartości `boolean` do funkcji jest naprawdę niepraktyczne. Od razu komplikuje to sygnaturę metody, wyraźnie informując, że funkcja wykonuje więcej niż jedną operację. Wykonuje jedną operację, jeżeli znacznik ma wartość `true`, i inną, jeżeli znacznik ma wartość `false`!

W funkcji z listingu 3.7 nie miałem wyboru, ponieważ wywołujący przesyłali już wcześniej znacznik, a ja chciałem ograniczyć zakres przebudowy do tej funkcji i kolejnych. Nadal funkcja ta jest wywoływana jako `render(true)`, co po prostu myli biednego czytelnika. Zobaczenie sygnatury funkcji, `render(boolean isSuite)`, niewiele pomaga. Powinniśmy podzielić tę funkcję na dwie: `renderForSuite()` oraz `renderForSingleTest()`.

Funkcje dwuargumentowe

Funkcja z dwoma argumentami jest trudniejsza do zrozumienia niż jednoargumentowa. Na przykład `writeField(name)` jest łatwiejsza do zrozumienia niż `writeField(output-Stream, name)`[10]. Mimo że znaczenie obu jest jasne, tylko pierwsza wyraźnie informuje o swoim przeznaczeniu. Druga wymaga krótkiego zastanowienia (trzeba zignorować pierwszy parametr). A *to* oczywiście w końcu spowoduje powstanie problemów — nigdy nie powinniśmy ignorować żadnej części kodu. W ignorowanych przez nas częściach zwykle kryją się błędy.

Istnieją oczywiście przypadki, w których dwa argumenty są odpowiednie. Na przykład `Point p = new Point(0,0);` jest zupełnie sensowne. Punkty kartezjańskie naturalnie korzystają z dwóch argumentów. W rzeczywistości byłbym bardzo zaskoczony, jeżeli zobaczyłbym wywołanie `new Point(0)`.

[10] Właśnie skończyłem przebudowę modułu, który korzystał z funkcji dwuargumentowych. Byłem w stanie zdefiniować `outputString` jako pole klasy i zamienić wszystkie wywołania `writeField` na jednoargumentowe. Wynik był znacznie bardziej czytelny.

Jednak dwa argumenty są w tym przypadku *uporządkowanymi składnikami jednej wartości*! Z kolei `outputStream` oraz `name` nie są naturalnie skojarzone ani nie mają naturalnego uporządkowania.

Nawet oczywiste funkcje dwuargumentowe, takie jak `assertEquals(expected, actual)`, są problematyczne. Ile razy umieściłeś wartość w `actual`, a powinna ona znajdować się w `expected`? Te dwa argumenty nie mają naturalnego uporządkowania. Uporządkowanie `expected`, `actual` jest konwencją, która wymaga nauki i praktyki.

Funkcje dwuargumentowe nie są złe i oczywiście trzeba je pisać. Jednak powinniśmy wiedzieć, że są one kosztowne, i znać dostępne mechanizmy pozwalające na ich konwersję na jednoargumentowe. Na przykład można przenieść metodę `writeField` do klasy `outputStream`, co upraszcza wywołanie do `outputStream.writeField(name)`. Można również umieścić `outputStream` w zmiennej składowej bieżącej klasy, dzięki czemu nie trzeba jej przekazywać. Można też wyodrębnić nową klasę, na przykład `FieldWriter`, która oczekuje `outputStream` w konstruktorze i posiada metodę `write`.

Funkcje trzyargumentowe

Funkcje oczekujące trzech argumentów są znacznie trudniejsze do zrozumienia niż dwuargumentowe. Problemy z kolejnością i ignorowaniem są znacznie większe. Warto dogłębnie przemyśleć problem przed utworzeniem funkcji trzyargumentowej.

Jako przykład weźmy często stosowaną przeciążoną wersję `assertEquals`, która oczekuje trzech argumentów: `assertEquals(message, expected, actual)`. Ile razy odczytałeś `message`, myśląc, że jest to wartość `expected`? Osobiście wiele razy potykałem się na tej właśnie funkcji trzyargumentowej. W rzeczywistości za każdym razem, gdy ją widzę, muszę uczyć się ignorować parametr `message`.

Z drugiej strony, istnieje funkcja trzyargumentowa, która nie jest tak podstępna: `assertEquals(1.0, amount, .001)`. Choć nadal wymaga ona dłuższego zastanowienia, to jednak warto o niej wspomnieć. Zawsze trzeba pamiętać, że równość wartości zmiennoprzecinkowych jest rzeczą względną.

Argumenty obiektowe

Gdy wydaje się, że funkcja wymaga więcej niż dwóch lub trzech argumentów, najprawdopodobniej niektóre z nich powinny być umieszczone w osobnej klasie. Przeanalizujmy różnice pomiędzy następującymi deklaracjami:

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Zmniejszenie liczby argumentów przez utworzenie z nich obiektów może wydawać się oszustwem, ale tak nie jest. Gdy grupy zmiennych są przesyłane wspólnie, tak jak `x` oraz `y` w powyższym przykładzie, najprawdopodobniej są częścią obiektu, który zasługuje na własną nazwę.

Listy argumentów

Czasami chcemy przekazać do funkcji różną liczbę argumentów. Jako przykład weźmy metodę `String.format`:

```
String.format("%s pracował %.2f godzin.", name, hours);
```

Jeżeli wszystkie argumenty będą traktowane w jednakowy sposób, tak jak w powyższym przykładzie, to są one odpowiednikiem jednego argumentu typu `List`. Stosując takie wnioskowanie, można powiedzieć, że `String.format` jest faktycznie metodą dwuargumentową. W rzeczywistości zamieszczona poniżej deklaracja `String.format` jest dwuargumentowa.

```
public String format(String format, Object... args)
```

Tak więc obowiązują te same reguły. Funkcje z argumentami mogą być jednoargumentowe, dwuargumentowe, a nawet trzyargumentowe. Jednak stosowanie większej liczby argumentów jest pomyłką.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

Czasowniki i słowa kluczowe

Odpowiednia nazwa funkcji może w znacznym stopniu wyjaśniać jej przeznaczenie oraz porządek i przeznaczenie argumentów. W przypadku funkcji jednoargumentowych funkcja i argument mogą tworzyć użyteczną parę czasownik-rzeczownik. Na przykład zapis `write(name)` jest oczywisty. Niezależnie od tego, czym jest `name` (nazwa), jest to poddawane operacji `write` (zapis). Jeszcze lepszą nazwą może być `writeField(name)`, z której jednoznacznie wynika, że zapisywana jest nazwa pola.

Jest to przykład nazwy funkcji ze *słowem kluczowym*. Przy użyciu tej postaci kodujemy nazwy argumentów w nazwie funkcji. Na przykład `assertEquals` może być lepiej nazwane jako `assertExpectedEqualsActual(expected, actual)`. W znacznym stopniu ogranicza to problem konieczności pamiętania kolejności argumentów.

Unikanie efektów ubocznych

Efekty uboczne są kłamstwem. Funkcja obiecuje, że wykonuje jedną operację, ale oprócz tego realizuje inną w sposób *ukryty*. Czasami w niespodziewany sposób modyfikuje zmienną ze swojej klasy. Czasami zmienia parametry przekazane do funkcji lub globalne zmienne systemowe. W obu przypadkach są to nieodpowiedzialne i niszczące operacje, które często powodują sprzężenia i zależności czasowe.

Jako przykład weźmy pozornie niewinną funkcję z listingu 3.6. Korzysta ona ze standardowego algorytmu do dopasowania nazwy użytkownika i jego hasła. Zwraca ona `true`, jeżeli podane wartości pasują, i `false`, jeżeli coś pójdzie źle. Ma ona również efekt uboczny. Czy możesz go znaleźć?

LISTING 3.6. UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Hasło prawidłowe".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

Efektem ubocznym jest oczywiście wywołanie `Session.initialize()`. Funkcja `checkPassword`, zgodnie ze swoją nazwą, powinna sprawdzać hasło. Nazwa nie informuje o tym, że jest inicjalizowana sesja. Wywołujący, który wierzy w to, co mówi nazwa funkcji, naraża się na ryzyko usunięcia istniejących danych sesji przy sprawdzeniu hasła użytkownika.

Ten efekt uboczny wywołuje sprzężenie czasowe. Powoduje ono, że `checkPassword` może być wywołana w określonym momencie (inaczej mówiąc, gdy można bezpiecznie zainicjować sesję). Jeżeli jest wywołana w niewłaściwym momencie, wszystkie dane sesji zostaną utracone. Sprzężenia czasowe są mylące, szczególnie jeżeli są ukryte w efektach ubocznych. Jeżeli konieczne jest użycie sprzężenia czasowego, należy jasno to przedstawić w nazwie funkcji. W tym przypadku wymaga to zmiany nazwy funkcji na `checkPasswordAndInitializeSession`, choć powoduje to naruszenie zasady wykonywania jednej operacji.

Argumenty wyjściowe

Argumenty są w naturalny sposób interpretowane jako dane *wejściowe* do funkcji. Jeżeli ktoś programuje więcej niż kilka lat, to jestem pewien, że dwa razy dłużej zajmuje mu zrozumienie argumentów *wyjściowych*. Na przykład:

```
appendFooter(s);
```

Czy ta funkcja dołącza `s` jako stopkę do czegoś? Czy dołącza stopkę do `s`? Czy `s` jest wejściem, czy wyjściem? Konieczne jest szybkie sprawdzenie sygnatury funkcji:

```
public void appendFooter(StringBuffer report)
```

Pozwala to rozwiązać problem, ale tylko kosztem sprawdzenia deklaracji funkcji. Wszystko, co wymusza na nas sprawdzenie sygnatury funkcji, jest odpowiednikiem dwukrotnego sprawdzania. Jest to przerwa w rozumowaniu, której należy unikać.

W czasach przed programowaniem obiektowym nieraz niezbędne było korzystanie z argumentów wyjściowych. Jednak większość przyczyn wykorzystywania argumentów wyjściowych zniknęła w językach obiektowych, ponieważ zmienna `this` jest *przewidziana* do tego, aby działała jak

argument wyjściowy. Inaczej mówiąc, znacznie lepiej byłoby, aby funkcja `appendFooter` była wywoływana jako

```
report.appendFooter();
```

Zwykle daje się uniknąć stosowania parametrów wyjściowych. Jeżeli funkcja musi zmieniać stan czegokolwiek, powinna zmieniać stan własnego obiektu.

Rozdzielanie poleceń i zapytań

Funkcje powinny coś wykonywać lub odpowiadać na jakieś pytanie, ale nie powinny robić tych dwóch operacji jednocześnie. Nasza funkcja powinna zmieniać stan obiektu lub zwracać pewne informacje na temat tego obiektu. Wykonywanie obu tych operacji często prowadzi do pomyłek. Jako przykład weźmy następującą funkcję:

```
public boolean set(String attribute, String value);
```

Ustawia ona wartość nazwanego atrybutu i zwraca `true` w przypadku powodzenia i `false`, jeżeli taki atrybut nie istnieje. Powoduje to powstanie dziwnych instrukcji, takich jak:

```
if (set("username", "unclebob"))...
```

Przeanalizujmy je z punktu widzenia czytelnika. Co to oznacza? Czy sprawdzamy, czy atrybut `username` miał wcześniej wartość `unclebob`? Czy też sprawdzamy, czy udało się ustawić atrybut `username` na wartość `unclebob`? Trudno wywnioskować znaczenie z wywołania, ponieważ nie jest jasne, czy słowo `set` jest czasownikiem, czy przymiotnikiem.

Autor miał zamiar, aby `set` było czasownikiem, ale w kontekście instrukcji `if` *mamy wrażenie*, że jest to przymiotnik. Z tego powodu czytamy ten kod następująco: „jeżeli atrybut `username` miał wcześniej wartość `unclebob`", a nie „ustawiamy wartość atrybutu `username` na `unclebob` i jeżeli to zadziała, to…". Możemy spróbować rozwiązać problem przez zmianę nazwy funkcji `set` na `setAndCheckIfExists`, ale nie poprawia to czytelności instrukcji `if`. Lepszym rozwiązaniem jest oddzielenie polecenia od zapytania, dzięki czemu niejasność nie występuje.

```
if (attributeExists("username")) {
   setAttribute("username", "unclebob");
   ...
}
```

Stosowanie wyjątków zamiast zwracania kodów błędów

Zwracanie kodów błędów z funkcji poleceń jest subtelnym naruszeniem rozdzielenia poleceń i zapytań. Promuje to stosowanie poleceń jako wyrażeń w warunkach instrukcji `if`.

```
if (deletePage(page) == E_OK)
```

Nie powoduje to problemów związanych z myleniem czasownika z przymiotnikiem, ale prowadzi do powstania głęboko zagnieżdżonych struktur. Gdy zwracamy kod błędu, powodujemy konieczność jego natychmiastowej obsługi przez wywołującego.

```
if (deletePage(page) == E_OK) {
   if (registry.deleteReference(page.name) == E_OK) {
      if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
         logger.log("strona usunięta");
      } else {
         logger.log("configKey nie został usunięty");
      }
   } else {
      logger.log("deleteReference w rejestrze nieudane");
   }
} else {
   logger.log("usuwanie nieudane");
   return E_ERROR;
}
```

Z drugiej strony, jeżeli użyjemy wyjątków zamiast zwracania kodów błędów, to przetwarzanie błędu może być oddzielone od prawidłowej ścieżki wykonania kodu i przez to uproszczone:

```
try {
   deletePage(page);
   registry.deleteReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
   logger.log(e.getMessage());
}
```

Wyodrębnienie bloków try-catch

Bloki `try-catch` naruszają strukturę kodu i mieszają przetwarzanie błędów ze zwykłym przetwarzaniem. Z tego powodu warto wyodrębniać treść bloków `try` i `catch` do osobnych funkcji.

```
public void delete(Page page) {
   try {
      deletePageAndAllReferences(page);
   }
   catch (Exception e) {
      logError(e);
   }
}

private void deletePageAndAllReferences(Page page) throws Exception {
   deletePage(page);
   registry.deteleReference(page.name);
   configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e){
   logger.log(e.getMessage());
}
```

W powyższym przypadku funkcja `delete` jest przeznaczona do obsługi błędów. Łatwo ją zrozumieć, a następnie zignorować. Funkcja `deletePageAndAllReferences` jest odpowiedzialna za przetwarzanie pełnego usuwania obiektu `page`. Obsługa błędów może być zignorowana. Zapewnia to elegancką separację, która pozwala na łatwiejsze zrozumienie i modyfikowanie kodu.

Obsługa błędów jest jedną operacją

Funkcje powinny wykonywać jedną operację. Obsługa błędów jest jedną operacją. Dlatego funkcja obsługi błędów nie powinna wykonywać nic innego. Powoduje to (jak w powyższym przykładzie), że jeżeli w funkcji istnieje słowo kluczowe, `try` powinno być pierwszym słowem w funkcji i nie powinno się w niej znajdować nic poza blokami `catch` i `finally`.

Przyciąganie zależności w Error.java

Zwracanie błędów zwykle powoduje, że istnieje wiele klas lub typów wyliczeniowych, w których są zdefiniowane kody błędów.

```
public enum Error {
   OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES,
   WAITING_FOR_EVENT;
}
```

Tego typu klasy są *magnesami zależności* — wiele klas musi je importować i z nich korzystać. Dlatego w momencie zmiany typu wyliczeniowego `Error` wszystkie te klasy muszą być ponownie skompilowane i zainstalowane[11]. Skutkuje to tym, że programiści nie chcą dodawać nowych błędów, ponieważ muszą wszystko przekompilować i instalować. Dlatego korzystają ze starych kodów błędów, zamiast dodawać nowe.

Gdy korzystamy z wyjątków zamiast kodów błędów, nowe wyjątki *dziedziczą* po klasie wyjątku. Mogą być one dodawane bez konieczności ponownej kompilacji i instalacji[12].

Nie powtarzaj się[13]

Gdy uważnie przeanalizujemy kod z listingu 3.1, zauważymy, że jest to algorytm powtarzający się cztery razy, kolejno dla przypadków `SetUp`, `SuiteSetUp`, `TearDown` oraz `SuiteTearDown`. Niełatwo zauważyć to powtórzenie, ponieważ te cztery przypadki są wymieszane z innym kodem i nie są powielone w identyczny sposób. Powtarzanie to jest problemem, ponieważ powoduje rozdęcie kodu i wymaga czterokrotnej modyfikacji w przypadku jakiejkolwiek zmiany algorytmu, a zatem czterokrotnie zwiększa możliwość popełnienia błędu.

[11] Ci, którzy uważali, że da się to zrobić bez ponownej kompilacji i instalacji, zostali odnalezieni — i poradziliśmy sobie z nimi.

[12] Jest to przykład zasady otwarty-zamknięty (OCP) [PPP02].

[13] Zasada DRY [PRAG].

Powielanie to zostało rozwiązane przy użyciu metody `include` z listingu 3.7. Warto przeczytać ponownie kod — z pewnością łatwo można zauważyć, że dzięki ograniczeniu powtórek wzrosła czytelność modułu.

Powtórzenia mogą być źródłem wszelkiego zła w oprogramowaniu. W celu ich kontrolowania i eliminowania powstało wiele zasad i praktyk. Jako przykład weźmy pod uwagę wszystkie postacie normalne bazy danych zdefiniowane przez Codda, które służą eliminowaniu danych. Weźmy pod uwagę sposób, w jaki w programowaniu obiektowym koncentrujemy kod w klasach bazowych, który w przeciwnym razie musiałby być nadmiarowy. Programowanie strukturalne, programowanie aspektowe, programowanie komponentowe — są po części strategiami eliminowania powtórzeń. Może się wydawać, że od czasów wynalezienia podprogramów innowacje w tworzeniu oprogramowania są kolejnymi próbami eliminowania powtórzeń w kodzie źródłowym.

Programowanie strukturalne

Niektórzy programiści korzystają z zasad programowania strukturalnego, opracowanych przez Edserera Dijkstrę[14]. Dijkstra powiedział, że każda funkcja i każdy blok funkcji powinien mieć jedno wejście i jedno wyjście. Zgodnie z tymi zasadami w funkcji powinna być jedna instrukcja `return`, w pętli nie należy stosować elementów `break` ani `continue` i *nigdy*, przenigdy nie wolno korzystać z instrukcji `goto`.

Choć sympatyzujemy z celami i dyscypliną programowania strukturalnego, zasady te niewiele wnoszą, jeżeli funkcje są małe. Zasady te są przydatne wyłącznie w dużych funkcjach.

Jeżeli więc nasze funkcje będą małe, to okazjonalne wielokrotne instrukcje `return`, `break` i `continue` nie powodują problemów, a czasami mogą przysłużyć się bardziej niż zasada jednego wejścia i jednego wyjścia. Z drugiej strony, instrukcja `goto` jest sensowna jedynie w dużych funkcjach, więc powinniśmy jej unikać.

Jak pisać takie funkcje?

Pisanie oprogramowania jest podobne do innych rodzajów pisania. Gdy piszemy książkę lub artykuł, najpierw zapisujemy swoje myśli, a następnie cyzelujemy je, by były czytelne i zrozumiałe. Pierwszy szkic może być zagmatwany i mało zorganizowany, więc pracujemy nad słowami, przebudowujemy i dopracowujemy tekst do momentu, aż będzie nadawał się do czytania.

Gdy piszę funkcję, pierwsza wersja jest zwykle długa i skomplikowana. Ma wiele wcięć i zagnieżdżonych pętli. Ma również długą listę argumentów. Nazwy są dowolne, a w funkcji znajduje się dużo powielonego kodu. Jednak mam również zbiór testów jednostkowych pokrywających każdy z tych zagmatwanych wierszy kodu.

---

[14] [SP72].

Zaczynam więc pracować nad kodem, wydzielać funkcje, zmieniać nazwy i eliminować powtórzenia. Zmniejszam metody i zmieniam ich kolejność. Czasami nawet wyodrębniam całe klasy, stale zapewniając poprawność wykonania testów.

Na końcu otrzymuję funkcje, które spełniają zasady przedstawione w tym rozdziale. Przedstawiona metoda jest może mało atrakcyjna, ale z pewnością skuteczna.

Zakończenie

Każdy system jest budowany na podstawie języka specyficznego dla domeny, zaprojektowanego przez programistów opisujących system. Funkcje są czasownikami języka, a klasy jego rzeczownikami. Nie jest przypadkiem, że w od dawna stosowanej metodologii rzeczowniki i czasowniki w dokumentach wymagań są pierwszymi kandydatami do klas i funkcji w systemie. Tak — sztuka programowania jest i zawsze była sztuką projektowania języków.

Świetni programiści postrzegają systemy jako opowiadania, a nie jako programy do napisania. Korzystają oni z możliwości wybranego języka programowania w celu skonstruowania znacznie bogatszego języka, który może być użyty do opowiedzenia danej historii. Częścią tego języka specyficznego dla domeny jest hierarchia funkcji opisujących wszystkie akcje, jakie są wykonywane w systemie. Akcje te są w sposób rekurencyjny zapisywane przy użyciu definiowanego języka specyficznego dla domeny, aby było możliwe opowiedzenie własnej części historii.

W niniejszym rozdziale przedstawiona została również mechanika pisania funkcji. Jeżeli będziemy stosować przedstawione tu zasady, funkcje będą krótkie, odpowiednio nazwane i ładnie zorganizowane. Nigdy nie należy zapominać, że naszym celem jest opowiadanie historii na temat systemu i pisane przez nas funkcje muszą tworzyć jasny i precyzyjny język pomagający nam w tym opowiadaniu.

SetupTeardownIncluder

LISTING 3.7. SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;

  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }
```

```
  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }

  private void includeSetupAndTeardownPages() throws Exception {
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updatePageContent();
  }

  private void includeSetupPages() throws Exception {
    if (isSuite)
      includeSuiteSetupPage();
    includeSetupPage();
  }

  private void includeSuiteSetupPage() throws Exception {
    include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
  }

  private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");
  }

  private void includePageContent() throws Exception {
    newPageContent.append(pageData.getContent());
  }

  private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if (isSuite)
      includeSuiteTeardownPage();
  }

  private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");
  }

  private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
  }

  private void updatePageContent() throws Exception {
    pageData.setContent(newPageContent.toString());
  }

  private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
```

```
        String pagePathName = getPathNameForPage(inheritedPage);
        buildIncludeDirective(pagePathName, arg);
      }
    }

    private WikiPage findInheritedPage(String pageName) throws Exception {
      return PageCrawlerImpl.getInheritedPage(pageName, testPage);
    }

    private String getPathNameForPage(WikiPage page) throws Exception {
      WikiPagePath pagePath = pageCrawler.getFullPath(page);
      return PathParser.render(pagePath);
    }

    private void buildIncludeDirective(String pagePathName, String arg) {
      newPageContent
        .append("\n!include ")
        .append(arg)
        .append(" .")
        .append(pagePathName)
        .append("\n");
    }
  }
```

Bibliografia

[KP78]: Kernighan i Plaugher, *The Elements of Programming Style*, McGraw-Hill 1978.

[PPP02]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall 2002.

[GOF]: Gamma, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

[PRAG]: Andrew Hunt, Dave Thomas, *The Pragmatic Programmer*, Addison-Wesley 2000.

[SP72]: O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press, London 1972.

Komentarze

Nie komentuj złego kodu — popraw go.

Brian W. Kernighan i P.J. Plaugher[1]

NIEWIELE JEST RZECZY TAK POMOCNYCH, jak dobrze umieszczony komentarz. Jednocześnie nic tak nie zaciemnia modułu, jak kilka zbyt dogmatycznych komentarzy. Nic nie jest tak szkodliwe, jak stary komentarz szerzący kłamstwa i dezinformację.

Komentarze nie są jak „Lista Schindlera". Nie są one „czystym dobrem". W rzeczywistości komentarze są w najlepszym przypadku koniecznym złem. Jeżeli nasz język programowania jest wystarczająco ekspresyjny lub mamy wystarczający talent, by wykorzystywać ten język, aby wyrażać nasze zamierzenia, nie będziemy potrzebować zbyt wielu komentarzy.

---

[1] [KP78], s. 144.

Prawidłowe zastosowanie komentarzy jest kompensowaniem naszych błędów przy tworzeniu kodu. Proszę zwrócić uwagę, że użyłem słowa *błąd*. Dokładnie to miałem na myśli. Obecność komentarzy zawsze sygnalizuje nieporadność programisty. Musimy korzystać z nich, ponieważ nie zawsze wiemy, jak wyrazić nasze intencje bez ich użycia, ale ich obecność nie jest powodem do świętowania.

Gdy uznamy, że konieczne jest napisanie komentarza, należy pomyśleć, czy nie istnieje sposób na wyrażenie tego samego w kodzie. Za każdym razem, gdy wyrazimy to samo za pomocą kodu, powinniśmy odczuwać satysfakcję. Za każdym razem, gdy piszemy komentarz, powinniśmy poczuć smak porażki.

Dlaczego jestem tak przeciwny komentarzom? Ponieważ one kłamią. Nie zawsze, nie rozmyślnie, ale nader często. Im starsze są komentarze, tym większe prawdopodobieństwo, że są po prostu błędne. Powód jest prosty. Programiści nie są w stanie utrzymywać ich aktualności.

Kod zmienia się i ewoluuje. Jego fragmenty są przenoszone w różne miejsca. Fragmenty te są rozdzielane, odtwarzane i ponownie łączone. Niestety, komentarze nie zawsze za nimi podążają — nie zawsze *mogą* być przenoszone. Zbyt często komentarze są odłączane od kodu, który opisują, i stają się osieroconymi notatkami o stale zmniejszającej się dokładności. Dla przykładu warto spojrzeć, co się stało z komentarzem i wierszem, którego dotyczył:

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
   "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
   "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Przykład: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Pozostałe zmienne instancyjne zostały prawdopodobnie później dodane pomiędzy stałą `HTTP_DATE_REGEXP` a objaśniającym ją komentarzem.

Można oczywiście stwierdzić, że programiści powinni być na tyle zdyscyplinowani, aby utrzymywać komentarze w należytym stanie. Zgadzam się, powinni. Wolałbym jednak, aby poświęcona na to energia została spożytkowana na zapewnienie takiej precyzji i wyrazistości kodu, by komentarze okazały się zbędne.

Niedokładne komentarze są znacznie gorsze niż ich brak. Kłamią i wprowadzają w błąd. Powodują powstanie oczekiwań, które nigdy nie są spełnione. Definiują stare zasady, które nie są już potrzebne lub nie powinny być stosowane.

Prawda znajduje się w jednym miejscu: w kodzie. Jedynie kod może niezawodnie przedstawić to, co realizuje. Jest jedynym źródłem naprawdę dokładnych informacji. Dlatego choć komentarze są czasami niezbędne, poświęcimy sporą ilość energii na zminimalizowanie ich liczby.

Komentarze nie są szminką dla złego kodu

Jednym z często spotykanych powodów pisania komentarzy jest nieudany kod. Napisaliśmy moduł i zauważamy, że jest źle zorganizowany. Wiemy, że jest chaotyczny. Mówimy wówczas: „Hm, będzie lepiej, jak go skomentuję". Nie! Lepiej go poprawić!

Precyzyjny i czytelny kod z małą liczbą komentarzy jest o wiele lepszy niż zabałaganiony i złożony kod z mnóstwem komentarzy. Zamiast spędzać czas na pisaniu kodu wyjaśniającego bałagan, jaki zrobiliśmy, warto poświęcić czas na posprzątanie tego bałaganu.

Czytelny kod nie wymaga komentarzy

W wielu przypadkach kod mógłby zupełnie obejść się bez komentarzy, a jednak programiści wolą umieścić w nim komentarz, zamiast zawrzeć objaśnienia w samym kodzie. Spójrzmy na poniższy przykład. Co wolelibyśmy zobaczyć? To:

```
// Sprawdzenie, czy pracownik ma prawo do wszystkich korzyści
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

czy to:

```
if (employee.isEligibleForFullBenefits())
```

Przeznaczenie tego kodu jest jasne po kilku sekundach myślenia. W wielu przypadkach jest to wyłącznie kwestia utworzenia funkcji, która wyraża to samo co komentarz, jaki chcemy napisać.

Dobre komentarze

Czasami komentarze są niezbędne lub bardzo przydatne. Przedstawimy kilka przypadków, w których uznaliśmy, że warto poświęcić im czas. Należy jednak pamiętać, że naprawdę dobry komentarz to taki, dla którego znaleźliśmy powód, aby go nie pisać.

Komentarze prawne

Korporacyjne standardy kodowania czasami wymuszają na nas pisanie pewnych komentarzy z powodów prawnych. Na przykład informacje o prawach autorskich są niezbędnym elementem umieszczanym w komentarzu na początku każdego pliku źródłowego.

Przykładem może być standardowy komentarz, jaki umieszczaliśmy na początku każdego pliku źródłowego w FitNesse. Na szczęście nasze środowisko IDE ukrywa te komentarze przez ich automatyczne zwinięcie.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Tego typu komentarze nie powinny być wielkości umów lub kodeksów. Tam, gdzie to możliwe, warto odwoływać się do standardowych licencji lub zewnętrznych dokumentów, a nie umieszczać w komentarzu wszystkich zasad i warunków.

Komentarze informacyjne

Czasami przydatne jest umieszczenie w komentarzu podstawowych informacji. Na przykład w poniższym komentarzu objaśniamy wartość zwracaną przez metodę abstrakcyjną.

```
// Zwraca testowany obiekt Responder.
protected abstract Responder responderInstance();
```

Komentarze tego typu są czasami przydatne, ale tam, gdzie to możliwe, lepiej jest skorzystać z nazwy funkcji do przekazania informacji. Na przykład w tym przypadku komentarz może stać się niepotrzebny, jeżeli zmienimy nazwę funkcji: `responderBeingTested`.

Poniżej mamy nieco lepszy przypadek:

```
// Dopasowywany format kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
   "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

W tym przypadku komentarze pozwalają nam poinformować, że użyte wyrażenie regularne ma dopasować czas i datę sformatowane za pomocą funkcji `SimpleDateFormat.format` z użyciem zdefiniowanego formatu. Nadal lepiej jest przenieść kod do specjalnej klasy pozwalającej na konwertowanie formatów daty i czasu. Po tej operacji komentarz najprawdopodobniej stanie się zbędny.

Wyjaśnianie zamierzeń

W niektórych przypadkach komentarze zawierają informacje nie tylko o implementacji, ale także o powodach podjęcia danej decyzji. W poniższym przypadku widzimy interesującą decyzję udokumentowaną w postaci komentarza. Przy porównywaniu obiektów autor zdecydował o tym, że obiekty jego klasy będą po posortowaniu wyżej niż obiekty pozostałych klas.

```
public int compareTo(Object o)
{
   if(o instanceof WikiPagePath)
   {
      WikiPagePath p = (WikiPagePath) o;
      String compressedName = StringUtil.join(names, "");
      String compressedArgumentName = StringUtil.join(p.names, "");
      return compressedName.compareTo(compressedArgumentName);
   }
   return 1; // Jesteśmy więksi, ponieważ jesteśmy właściwego typu.
}
```

Poniżej pokazany jest lepszy przykład. Możemy nie zgadzać się z rozwiązaniem tego problemu przez programistę, ale przynajmniej wiemy, co próbował zrobić.

```
public void testConcurrentAddWidgets() throws Exception {
   WidgetBuilder widgetBuilder =
   new WidgetBuilder(new Class[]{BoldWidget.class});
   String text = "'''bold text'''";
```

```
    ParentWidget parent =
       new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);

    //Jest to nasza próba uzyskania wyścigu
    //przez utworzenie dużej liczby wątków.
    for (int i = 0; i < 25000; i++) {
       WidgetBuilderThread widgetBuilderThread =
       new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
       Thread thread = new Thread(widgetBuilderThread);
       thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

Wyjaśnianie

Czasami przydatne jest wytłumaczenie znaczenia niejasnych argumentów lub zwracanych wartości. Zwykle lepiej jest znaleźć sposób na to, by ten argument lub zwracana wartość były bardziej czytelne, ale jeżeli są one częścią biblioteki standardowej lub kodu, którego nie możemy zmieniać, to wyjaśnienia w komentarzach mogą być użyteczne.

```
public void testCompareTo() throws Exception
{
   WikiPagePath a = PathParser.parse("PageA");
   WikiPagePath ab = PathParser.parse("PageA.PageB");
   WikiPagePath b = PathParser.parse("PageB");
   WikiPagePath aa = PathParser.parse("PageA.PageA");
   WikiPagePath bb = PathParser.parse("PageB.PageB");
   WikiPagePath ba = PathParser.parse("PageB.PageA");

   assertTrue(a.compareTo(a) == 0); // a == a
   assertTrue(a.compareTo(b) != 0); // a != b
   assertTrue(ab.compareTo(ab) == 0); // ab == ab
   assertTrue(a.compareTo(b) == -1); // a < b
   assertTrue(aa.compareTo(ab) == -1); // aa < ab
   assertTrue(ba.compareTo(bb) == -1); // ba < bb
   assertTrue(b.compareTo(a) == 1); // b > a
   assertTrue(ab.compareTo(aa) == 1); // ab > aa
   assertTrue(bb.compareTo(ba) == 1); // bb > ba
}
```

Istnieje oczywiście spore ryzyko, że komentarze objaśniające są nieprawidłowe. Warto przeanalizować poprzedni przykład i zobaczyć, jak trudno jest sprawdzić, czy są one prawidłowe. Wyjaśnia to, dlaczego niezbędne są objaśnienia i dlaczego są one ryzykowne. Tak więc przed napisaniem tego typu komentarzy należy sprawdzić, czy nie istnieje lepszy sposób, a następnie poświęcić im więcej uwagi, aby były precyzyjne.

Ostrzeżenia o konsekwencjach

Komentarze mogą również służyć do ostrzegania innych programistów o określonych konsekwencjach. Poniższy komentarz wyjaśnia, dlaczego przypadek testowy jest wyłączony:

```
// Nie uruchamiaj, chyba że masz nieco czasu do zagospodarowania.
public void _testWithReallyBigFile()
{
   writeLinesToFile(10000000);
   response.setBody(testFile);
   response.readyToSend(this);
   String responseString = output.toString();
   assertSubString("Content-Length:
   ↪1000000000", responseString);
   assertTrue(bytesSent > 1000000000);
}
```

Obecnie oczywiście wyłączamy przypadek testowy przez użycie atrybutu `@Ignore` z odpowiednim tekstem wyjaśniającym. `@Ignore("Zajmuje zbyt dużo czasu")`. Jednak w czasach przed JUnit 4 umieszczenie podkreślenia przed nazwą metody było często stosowaną konwencją. Komentarz, choć nonszalancki, dosyć dobrze wskazuje powód.

Poniżej pokazany jest inny przykład:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
   //SimpleDateFormat nie jest bezpieczna dla wątków,
   //więc musimy każdy obiekt tworzyć niezależnie.
   SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
   df.setTimeZone(TimeZone.getTimeZone("GMT"));
   return df;
}
```

Można narzekać, że istnieją lepsze sposoby rozwiązania tego problemu. Mogę się z tym zgodzić. Jednak zastosowany tu komentarz jest całkiem rozsądny. Może on powstrzymać nadgorliwego programistę przed użyciem statycznego inicjalizera dla zapewnienia lepszej wydajności.

Komentarze TODO

Czasami dobrym pomysłem jest pozostawianie notatek „do zrobienia” w postaci komentarzy `//TODO`. W zamieszczonym poniżej przypadku komentarz `TODO` wyjaśnia, dlaczego funkcja ma zdegenerowaną implementację i jaka powinna być jej przyszłość.

```
//TODO-MdM Nie jest potrzebna.
// Oczekujemy, że zostanie usunięta po pobraniu modelu.
protected VersionInfo makeVersion() throws Exception
{
   return null;
}
```

Komentarze TODO oznaczają zadania, które według programisty powinny być wykonane, ale z pewnego powodu nie można tego zrobić od razu. Może to być przypomnienie o konieczności usunięcia przestarzałej funkcji lub prośba do innej osoby o zajęcie się problemem. Może to być żądanie, aby ktoś pomyślał o nadaniu lepszej nazwy, lub przypomnienie o konieczności wprowadzenia zmiany zależnej od planowanego zdarzenia. Niezależnie od tego, czym jest TODO, *nie może* to być wymówka dla pozostawienia złego kodu w systemie.

Obecnie wiele dobrych IDE zapewnia specjalne funkcje lokalizujące wszystkie komentarze TODO, więc jest mało prawdopodobne, aby zostały zgubione. Nadal jednak nie jest korzystne, by kod był nafaszerowany komentarzami TODO. Należy więc regularnie je przeglądać i eliminować wszystkie, które się da.

Wzmocnienie

Komentarz może być użyty do wzmocnienia wagi operacji, która w przeciwnym razie może wydawać się niekonsekwencją.

```
String listItemContent = match.group(3).trim();
// Wywołanie trim jest naprawdę ważne. Usuwa początkowe
// spacje, które mogą spowodować, że element będzie
// rozpoznany jako kolejna lista.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Komentarze Javadoc w publicznym API

Nie ma nic bardziej pomocnego i satysfakcjonującego, jak dobrze opisane publiczne API. Przykładem tego może być standardowa biblioteka Java. Bez niej pisanie programów Java byłoby trudne, o ile nie niemożliwe.

Jeżeli piszemy publiczne API, to niezbędne jest napisanie dla niego dobrej dokumentacji Javadoc. Jednak należy pamiętać o pozostałych poradach z tego rozdziału. Komentarze Javadoc mogą być równie mylące, nie na miejscu i nieszczere jak wszystkie inne komentarze.

Złe komentarze

Do tej kategorii należy większość komentarzy. Zwykle są to podpory złego kodu lub wymówki albo uzasadnienie niewystarczających decyzji znaczące niewiele więcej niż dyskusja programisty ze sobą.

Bełkot

Pisanie komentarza tylko dlatego, że czujemy, iż powinien być napisany lub też że wymaga tego proces, jest błędem. Jeżeli decydujemy się na napisanie komentarza, musimy poświęcić nieco czasu na upewnienie się, że jest to najlepszy komentarz, jaki mogliśmy napisać.

Poniżej zamieszczony jest przykład znaleziony w FitNesse. Komentarz był faktycznie przydatny. Jednak autor śpieszył się lub nie poświęcił mu zbyt wiele uwagi. Bełkot, który po sobie zostawił, stanowi nie lada zagadkę:

```
public void loadProperties()
{
   try
   {
      String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
      FileInputStream propertiesStream = new FileInputStream(propertiesPath);
      loadedProperties.load(propertiesStream);
   }
   catch(IOException e)
   {
      // Brak plików właściwości oznacza załadowanie wszystkich wartości domyślnych.
   }
}
```

Co oznacza komentarz w bloku `catch`? Jasne jest, że znaczy on coś dla autora, ale znaczenie to nie zostało dobrze wyartykułowane. Jeżeli otrzymamy wyjątek `IOException`, najwyraźniej oznacza to brak pliku właściwości, a w takim przypadku ładowane są wszystkie wartości domyślne. Jednak kto ładuje te wartości domyślne? Czy były załadowane przed wywołaniem `loadProperties.load`? Czy też `loadProperties.load` przechwytuje wyjątek, ładuje wartości domyślne i przekazuje nam wyjątek do zignorowania? A może `loadProperties.load` ładuje wszystkie wartości domyślne przed próbą załadowania pliku? Czy autor próbował usprawiedliwić przed samym sobą fakt, że pozostawił pusty blok `catch`? Być może — ta możliwość jest nieco przerażająca — autor próbował powiedzieć sobie, że powinien wrócić w to miejsce i napisać kod ładujący wartości domyślne.

Jedynym sposobem, aby się tego dowiedzieć, jest przeanalizowanie kodu z innych części systemu i sprawdzenie, co się w nich dzieje. Wszystkie komentarze, które wymuszają zaglądanie do innych modułów w celu ich zrozumienia, nie są warte bitów, które zajmują.

Powtarzające się komentarze

Na listingu 4.1 zamieszczona jest prosta funkcja z komentarzem w nagłówku, który jest całkowicie zbędny. Prawdopodobnie dłużej zajmuje przeczytanie komentarza niż samego kodu.

LISTING 4.1. waitForClose

```
// Metoda użytkowa kończąca pracę, gdy this.closed ma wartość true. Zgłasza wyjątek,
// jeżeli przekroczony zostanie czas oczekiwania.
public synchronized void waitForClose(final long timeoutMillis)
  throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

Czemu służy ten komentarz? Przecież nie niesie więcej informacji niż sam kod. Nie uzasadnia on kodu, nie przedstawia zamierzeń ani przyczyn. Nie jest łatwiejszy do czytania od samego kodu.

W rzeczywistości jest mniej precyzyjny niż kod i wymusza na czytelniku zaakceptowanie braku precyzji w imię prawdziwego zrozumienia. Jest on podobny do paplania sprzedawcy używanych samochodów, który zapewnia, że nie musisz zaglądać pod maskę.

Spójrzmy teraz na legion bezużytecznych i nadmiarowych komentarzy Javadoc pobranych z programu Tomcat i zamieszczonych na listingu 4.2. Komentarze te mają za zadanie wyłącznie zaciemnić i popsuć kod. Nie mają one żadnej wartości dokumentującej. Co gorsza, pokazałem tutaj tylko kilka pierwszych. W tym module znajduje się znacznie więcej takich komentarzy.

LISTING 4.2. ContainerBase.java (Tomcat)

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * The processor delay for this component.
   */
  protected int backgroundProcessorDelay = -1;

  /**
   * The lifecycle event support for this component.
   */
  protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);

  /**
   * The container event listeners for this Container.
   */
  protected ArrayList listeners = new ArrayList();

  /**
   * The Loader implementation with which this Container is
   * associated.
   */
  protected Loader loader = null;

  /**
   * The Logger implementation with which this Container is
   * associated.
   */
  protected Log logger = null;

  /**
   * Associated logger name.
   */
  protected String logName = null;

  /**
   * The Manager implementation with which this Container is
   * associated.
   */
  protected Manager manager = null;

  /**
   * The cluster with which this Container is associated.
   */
  protected Cluster cluster = null;
```

```
/**
 * The human-readable name of this Container.
 */
protected String name = null;

/**
 * The parent Container to which this Container is a child.
 */
protected Container parent = null;

/**
 * The parent class loader to be configured when we install a
 * Loader.
 */
protected ClassLoader parentClassLoader = null;

/**
 * The Pipeline object with which this Container is
 * associated.
 */
protected Pipeline pipeline = new StandardPipeline(this);

/**
 * The Realm with which this Container is associated.
 */
protected Realm realm = null;

/**
 * The resources DirContext object with which this Container
 * is associated.
 */
protected DirContext resources = null;
```

Mylące komentarze

Czasami pomimo najlepszych intencji programista zapisuje w komentarzu nieprecyzyjne zdania. Wróćmy na moment do nadmiarowego, ale również nieco mylącego komentarza zamieszczonego na listingu 4.1.

Czy Czytelnik zauważył, w czym ten komentarz jest mylący? Metoda ta nie kończy się, *gdy* `this.closed` ma wartość `true`. Kończy się ona, *jeżeli* `this.closed` ma wartość `true`; w przeciwnym razie czeka określony czas, a następnie zgłasza wyjątek, *jeżeli* `this.closed` nadal nie ma wartości `true`.

Ta subtelna dezinformacja umieszczona w komentarzu, który czyta się trudniej niż sam kod, może spowodować, że inny programista naiwnie wywoła tę funkcję, oczekując, że zakończy się od razu, gdy `this.closed` przyjmie wartość `true`. Ten biedny programista może zorientować się, o co chodzi, dopiero w sesji debugera, gdy będzie próbował zorientować się, dlaczego jego kod działa tak powoli.

Komentarze wymagane

Wymaganie, aby każda funkcja posiadała Javadoc lub aby każda zmienna posiadała komentarz, jest po prostu głupie. Tego typu komentarze tylko zaciemniają kod i prowadzą do powszechnych pomyłek i dezorganizacji.

Na przykład wymaganie komentarza Javadoc prowadzi do powstania takich potworów, jak ten zamieszczony na listingu 4.3. Takie komentarze nie wnoszą niczego, za to utrudniają zrozumienie kodu.

LISTING 4.3.

```
/**
 *
 * @param title Tytuł płyty CD
 * @param author Autor płyty CD
 * @param tracks Liczba ścieżek na płycie CD
 * @param durationInMinutes Czas odtwarzania CD w minutach
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

Komentarze dziennika

Czasami programiści dodają na początku każdego pliku komentarz informujący o każdej edycji. Komentarze takie tworzą pewnego rodzaju dziennik wszystkich wprowadzonych zmian. Spotkałem się z modułami zawierającymi kilkanaście stron z kolejnymi pozycjami dziennika.

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package
* com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
* class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
* class is gone (DG); Changed getPreviousDayOfWeek(),
* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
* bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
* (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Dawno temu istniały powody tworzenia i utrzymywania takich dzienników na początku każdego modułu. Nie mieliśmy po prostu systemów kontroli wersji, które wykonywały to za nas. Obecnie jednak takie długie dzienniki tylko pogarszają czytelność modułu. Powinny zostać usunięte.

Komentarze wprowadzające szum informacyjny

Czasami zdarza się nam spotkać komentarze, które nie są niczym więcej jak tylko szumem informacyjnym. Przedstawiają one oczywiste dane i nie dostarczają żadnych nowych informacji.

```
/**
* Konstruktor domyślny.
*/
protected AnnualDateRule() {
}
```

No nie, *naprawdę*? Albo coś takiego:

```
/** Dzień miesiąca. */
private int dayOfMonth;
```

Następnie mamy doskonały przykład nadmiarowości:

```
/**
* Zwraca dzień miesiąca.
*
* @return dzień miesiąca.
*/
public int getDayOfMonth() {
   return dayOfMonth;
}
```

Komentarze takie stanowią tak duży szum informacyjny, że nauczyliśmy się je ignorować. Gdy czytamy kod, nasze oczy po prostu je pomijają. W końcu komentarze te głoszą nieprawdę, gdy otaczający kod jest zmieniany.

Pierwszy komentarz z listingu 4.4 wydaje się właściwy[2]. Wyjaśnia powód zignorowania bloku `catch`. Jednak drugi jest czystym szumem. Najwyraźniej programista był tak sfrustrowany pisaniem bloków `try-catch` w tej funkcji, że musiał sobie ulżyć.

LISTING 4.4. startSending

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // Normalne. Ktoś zatrzymał żądanie.
  }
  catch(Exception e)
```

[2] Obecny trend sprawdzania poprawności w komentarzach przez środowiska IDE jest zbawieniem dla wszystkich, którzy czytają dużo kodu.

```
    {
      try
      {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
      }
      catch(Exception e1)
      {
        // Muszę zrobić przerwę!
      }
    }
  }
```

Zamiast szukać ukojenia w bezużytecznych komentarzach, programista powinien zauważyć, że jego frustracja może być rozładowana przez poprawienie struktury kodu. Powinien skierować swoją energię na wyodrębnienie ostatniego bloku `try-catch` do osobnej funkcji, jak jest to pokazane na listingu 4.5.

LISTING 4.5. startSending (zmodyfikowany)

```
  private void startSending()
  {
    try
    {
      doSending();
    }
    catch(SocketException e)
    {
      // Normalne. Ktoś zatrzymał żądanie.
    }
    catch(Exception e)
    {
      addExceptionAndCloseResponse(e);
    }
  }

  private void addExceptionAndCloseResponse(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
    }
  }
```

Warto zastąpić pokusę tworzenia szumu determinacją do wyczyszczenia swojego kodu. Pozwala to stać się lepszym i szczęśliwszym programistą.

Przerażający szum

Komentarze Javadoc również mogą być szumem. Jakie jest przeznaczenie poniższych komentarzy Javadoc (ze znanej biblioteki open source)? Odpowiedź: żadne. Są to po prostu nadmiarowe komentarze stanowiące szum informacyjny, napisane w źle pojętej chęci zapewnienia dokumentacji.

```
/** Nazwa. */
private String name;
/** Wersja. */
private String version;
/** nazwaLicencji. */
private String licenceName;
/** Wersja. */
private String info;
```

Przeczytajmy dokładniej te komentarze. Czy czytelnik może zauważyć błąd kopiowania i wklejania? Jeżeli autor nie poświęcił uwagi pisaniu komentarzy (lub ich wklejaniu), to czy czytelnik może oczekiwać po nich jakiejś korzyści?

Nie używaj komentarzy, jeżeli można użyć funkcji lub zmiennej

Przeanalizujmy poniższy fragment kodu:

```
// Czy moduł z listy globalnej <mod> zależy
// od podsystemu, którego jest częścią?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Może to być przeorganizowane bez użycia komentarzy:

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Autor oryginalnego kodu prawdopodobnie napisał komentarz na początku (niestety), a następnie kod realizujący zadanie z komentarza. Jeżeli jednak autor zmodyfikowałby kod w sposób, w jaki ja to wykonałem, komentarz mógłby zostać usunięty.

Znaczniki pozycji

Czasami programiści lubią zaznaczać określone miejsca w pliku źródłowym. Na przykład ostatnio trafiłem na program, w którym znalazłem coś takiego:

```
// Akcje ////////////////////////////////
```

Istnieją rzadkie przypadki, w których sensowne jest zebranie określonych funkcji razem pod tego rodzaju transparentami. Jednak zwykle powodują one chaos, który powinien być wyeliminowany — szczególnie ten pociąg ukośników na końcu.

Transparent ten jest zaskakujący i oczywisty, jeżeli nie widzimy go zbyt często. Tak więc warto używać ich oszczędnie i tylko wtedy, gdy ich zalety są wyraźne. Jeżeli zbyt często używamy tych transparentów, zaczynają być traktowane jako szum tła i ignorowane.

Komentarze w klamrach zamykających

Zdarza się, że programiści umieszczają specjalne komentarze po klamrach zamykających, tak jak na listingu 4.6. Choć może to mieć sens w przypadku długich funkcji, z głęboko zagnieżdżonymi strukturami, w małych i hermetycznych funkcjach, jakie preferujemy, tworzą tylko nieład. Jeżeli więc Czytelnik będzie chciał oznaczać klamry zamykające, niech spróbuje zamiast tego skrócić funkcję.

LISTING 4.6. wc.java

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
      } //while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
    System.err.println("Error:" + e.getMessage());
    } //catch
  } //main
}
```

Atrybuty i dopiski

/ Dodane przez Ricka */*

Systemy kontroli wersji świetnie nadają się do zapamiętywania, kto (i kiedy) dodał określony fragment. Nie ma potrzeby zaśmiecania kodu tymi małymi dopiskami. Można uważać, że tego typu komentarze będą przydatne do sprawdzenia, z kim można porozmawiać na temat danego fragmentu kodu. Rzeczywistość jest inna — zwykle zostają tam przez lata, tracąc na dokładności i użyteczności.

Pamiętajmy — systemy kontroli wersji są lepszym miejscem dla tego rodzaju informacji.

Zakomentowany kod

Niewiele jest praktyk tak nieprofesjonalnych, jak zakomentowanie kodu. Nie rób tego!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Inni programiści, którzy zobaczą taki zakomentowany kod, nie będą mieli odwagi go usunąć. Uznają, że jest tam z jakiegoś powodu i że jest zbyt ważny, aby go usunąć. W ten sposób zakomentowany kod zaczyna się odkładać jak osad na dnie butelki zepsutego wina.

Przeanalizujmy fragment z projektu Apache:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
   writeEnd();
   this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
   this.pngBytes = null;
}
return this.pngBytes;
```

Dlaczego te dwa wiersze kodu są zakomentowane? Czy są ważne? Czy jest to pozostałość po wcześniejszych zmianach? Czy też są błędami, które ktoś przed laty zakomentował i nie zadał sobie trudu, aby to wyczyścić?

W latach sześćdziesiątych ubiegłego wieku komentowanie kodu mogło być przydatne. Jednak od bardzo długiego czasu mamy już dobre systemy kontroli wersji. Systemy te pamiętają za nas wcześniejszy kod. Nie musimy już komentować kodu. Po prostu możemy go usunąć. Nie stracimy go. Gwarantuję.

Komentarze HTML

Kod HTML w komentarzach do kodu źródłowego jest paskudny, o czym można się przekonać po przeczytaniu kodu zamieszczonego poniżej. Powoduje on, że komentarze są trudne do przeczytania w jedynym miejscu, gdzie powinny być łatwe do czytania — edytorze lub środowisku IDE. Jeżeli komentarze mają być pobierane przez jakieś narzędzie (na przykład Javadoc), aby mogły być wyświetlone na stronie WWW, to zadaniem tego narzędzia, a nie programisty, powinno być opatrzenie ich stosownymi znacznikami HTML.

```
/**
 * Zadanie uruchomienia testów sprawności.
 * Zadanie uruchamia testy fitnesse i publikuje wyniki.
 * <p/>
 * <pre>
 * Zastosowanie:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 * classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * LUB
 * &lt;taskdef classpathref=&quot;classpath&quot;
 * resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 * suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 * fitnesseport=&quot;8082&quot;
 * resultsdir=&quot;${results.dir}&quot;
 * resultshtmlpage=&quot;fit-results.html&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

Informacje nielokalne

Jeżeli konieczne jest napisanie komentarza, to należy upewnić się, że opisuje on kod znajdujący się w pobliżu. Nie należy udostępniać informacji dotyczących całego systemu w kontekście komentarzy lokalnych. Weźmy jako przykład zamieszczone poniżej komentarze Javadoc. Pomijając fakt, że są zupełnie zbędne, zawierają one informacje o domyślnym porcie. Funkcja jednak nie ma absolutnie żadnej kontroli nad tą wartością domyślną. Komentarz nie opisuje funkcji, ale inną część systemu, znacznie od niej oddaloną. Oczywiście, nie ma gwarancji, że komentarz ten zostanie zmieniony, gdy kod zawierający wartość domyślną ulegnie zmianie.

```
/**
 * Port, na którym działa fitnesse. Domyślnie <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
   this.fitnessePort = fitnessePort;
}
```

Nadmiar informacji

Nie należy umieszczać w komentarzach interesujących z punktu widzenia historii dyskusji lub luźnych opisów szczegółów. Komentarz zamieszczony poniżej został pobrany z modułu mającego za zadanie sprawdzić, czy funkcja może kodować i dekodować zgodnie ze standardem base64. Osoba czytająca ten kod nie musi znać wszystkich szczegółowych informacji znajdujących się w komentarzu, poza numerem RFC.

```
/*
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
Part One: Format of Internet Message Bodies
section 6.8. Base64 Content-Transfer-Encoding
The encoding process represents 24-bit groups of input bits as output
strings of 4 encoded characters. Proceeding from left to right,
a 24-bit input group is formed by concatenating 3 8-bit input groups.
These 24 bits are then treated as 4 concatenated 6-bit groups, each
of which is translated into a single digit in the base64 alphabet.
When encoding a bit stream via the base64 encoding, the bit stream
must be presumed to be ordered with the most-significant-bit first.
That is, the first bit in the stream will be the high-order bit in
the first 8-bit byte, and the eighth bit will be the low-order bit in
the first 8-bit byte, and so on.
*/
```

Nieoczywiste połączenia

Połączenie pomiędzy komentarzem a kodem, który on opisuje, powinno być oczywiste. Jeżeli mamy problemy z napisaniem komentarza, to powinniśmy przynajmniej doprowadzić do tego, by czytelnik patrzący na komentarz i kod rozumiał, o czym mówi dany komentarz.

Jako przykład weźmy komentarz zaczerpnięty z projektu Apache:

```
/*
* Zaczynamy od tablicy, która jest na tyle duża, aby zmieścić wszystkie piksele
* (plus filter bajtów) oraz dodatkowe 200 bajtów na informacje nagłówka.
*/
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Co to są bajty `filter`? Czy ma to jakiś związek z wyrażeniem `+1`? A może z `*3`? Z obydwoma? Czy piksel jest bajtem? Dlaczego 200? Zadaniem komentarza jest wyjaśnianie kodu, który sam się nie objaśnia. Jaka szkoda, że sam komentarz wymaga dodatkowego objaśnienia.

Nagłówki funkcji

Krótkie funkcje nie wymagają rozbudowanych opisów. Odpowiednio wybrana nazwa małej funkcji realizującej jedną operację jest zwykle lepsza niż nagłówek z komentarzem.

Komentarze Javadoc w niepublicznym kodzie

Komentarze Javadoc są przydatne w publicznym API, ale za to niemile widziane w kodzie nieprzeznaczonym do publicznego rozpowszechniania. Generowanie stron Javadoc dla klas i funkcji wewnątrz systemu zwykle nie jest przydatne, a dodatkowy formalizm komentarzy Javadoc przyczynia się jedynie do powstania błędów i rozproszenia uwagi.

Przykład

Kod zamieszczony na listingu 4.7 został przeze mnie napisany na potrzeby pierwszego kursu XP Immersion. Był on w zamierzeniach przykładem złego stylu kodowania i komentowania. Później Kent Beck przebudował go do znacznie przyjemniejszej postaci na oczach kilkudziesięciu entuzjastycznie reagujących studentów. Później zaadaptowałem ten przykład na potrzeby mojej książki *Agile Software Development, Principles, Patterns, and Practices* i pierwszych artykułów *Craftman* publikowanych w magazynie *Software Development*.

Fascynujące w tym module jest to, że swego czasu byłby on uznawany za „dobrze udokumentowany". Teraz postrzegamy go jako mały bałagan. Spójrzmy, jak wiele problemów z komentarzami można tutaj znaleźć.

LISTING 4.7. GeneratePrimes.java

```
/**
 * Klasa ta generuje liczby pierwsze do określonego przez użytkownika
 * maksimum. Użytym algorytmem jest sito Eratostenesa.
 * <p>
 * Eratostenes z Cyrene, urodzony 276 p.n.e. w Cyrene, Libia --
 * zmarł 194 p.n.e. w Aleksandrii. Pierwszy człowiek, który obliczył
 * obwód Ziemi. Znany również z prac nad kalendarzem
 * z latami przestępnymi i prowadzenia biblioteki w Aleksandrii.
 * <p>
 * Algorytm jest dosyć prosty. Mamy tablicę liczb całkowitych
 * zaczynających się od 2. Wykreślamy wszystkie wielokrotności 2. Szukamy
```

```
 * następnej niewykreślonej liczby i wykreślamy wszystkie jej wielokrotności.
 * Powtarzamy działania do momentu osiągnięcia pierwiastka kwadratowego z maksymalnej wartości.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
*/
import java.util.*;

  public class GeneratePrimes
  {
  /**
   * @param maxValue jest limitem generacji.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // Jedyny prawidłowy przypadek.
    {
      // Deklaracje.
      int s = maxValue + 1; // Rozmiar tablicy.
      boolean[] f = new boolean[s];
      int i;
      // Inicjalizacja tablicy wartościami true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // Usuwanie znanych liczb niebędących pierwszymi.
      f[0] = f[1] = false;

      // Sito.
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // Jeżeli i nie jest wykreślone, wykreślamy jego wielokrotności.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // Wielokrotności nie są pierwsze.
        }
      }

      // Ile mamy liczb pierwszych?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // Licznik trafień.
      }

      int[] primes = new int[count];

      // Przeniesienie liczb pierwszych do wyniku.
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i]) // Jeżeli pierwsza.
          primes[j++] = i;
      }
      return primes; // Zwracamy liczby pierwsze.
    }
    else // maxValue < 2
      return new int[0]; // Zwracamy pustą tablicę, jeżeli niewłaściwe dane wejściowe.
    }
}
```

Na listingu 4.8 zamieszczona jest przebudowana wersja tego samego modułu. Warto zauważyć, że znacznie ograniczona jest liczba komentarzy. W całym module znajdują się tylko dwa komentarze. Oba są z natury opisowe.

LISTING 4.8. PrimeGenerator.java (przebudowany)

```
/**
 * Klasa ta generuje liczby pierwsze do określonego przez użytkownika
 * maksimum. Użytym algorytmem jest sito Eratostenesa.
* Mamy tablicę liczb całkowitych zaczynających się od 2.
* Wyszukujemy pierwszą nieokreśloną liczbę i wykreślamy wszystkie jej
* wielokrotności. Powtarzamy, aż nie będzie więcej wielokrotności w tablicy.
*/

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Każda wielokrotność w tablicy ma podzielnik będący liczbą pierwszą
    // mniejszą lub równą pierwiastkowi kwadratowemu wielkości tablicy,
    // więc nie musimy wykreślać wielokrotności większych od tego pierwiastka.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
```

```
    }

    private static boolean notCrossed(int i)
    {
      return crossedOut[i] == false;
    }

    private static void putUncrossedIntegersIntoResult()
    {
      result = new int[numberOfUncrossedIntegers()];
      for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          result[j++] = i;
    }

    private static int numberOfUncrossedIntegers()
    {
      int count = 0;
      for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
          count++;

      return count;
    }
  }
```

Można się spierać, że pierwszy komentarz jest nadmiarowy, ponieważ czyta się go podobnie jak samą funkcję `genratePrimes`. Uważam jednak, że komentarz ułatwia czytelnikowi poznanie algorytmu, więc zdecydowałem o jego pozostawieniu.

Drugi komentarz jest niemal na pewno niezbędny. Wyjaśnia powody zastosowania pierwiastka jako ograniczenia pętli. Można sprawdzić, że żadna z prostych nazw zmiennych ani inna struktura kodu nie pozwala na wyjaśnienie tego punktu. Z drugiej strony, użycie pierwiastka może być próżnością. Czy faktycznie oszczędzam dużo czasu przez ograniczenie liczby iteracji do pierwiastka liczby? Czy obliczenie pierwiastka nie zajmuje więcej czasu, niż uda się nam zaoszczędzić?

Warto o tym pomyśleć. Zastosowanie pierwiastka jako limitu pętli zadowala siedzącego we mnie eksperta C i asemblera, ale nie jestem przekonany, że jest to warte czasu i energii osoby, która ma za zadanie zrozumieć ten kod.

Bibliografia

[KP78]: Kernighan i Plaugher, *The Elements of Programming Style*, McGraw-Hill 1978.

ROZDZIAŁ 5.

Formatowanie

GDY LUDZIE ZAGLĄDAJĄ POD MASKĘ PROJEKTU, chcą być oczarowani elegancją, spójnością i zauważalną dbałością o szczegóły. Chcemy ich zaskoczyć uporządkowaniem. Chcemy, aby ich brwi się uniosły, gdy będą przeglądać moduły kodu. Chcemy, aby rozpoznali pracę profesjonalisty. Jeżeli jednak zobaczą wymieszaną masę kodu, która wygląda, jakby kod był pisany przez załogę pijanych marynarzy, mogą dojść do wniosku, że taki sam brak dbałości o szczegóły dotyczy innych aspektów projektu.

Pamiętajmy, że kod powinien być ładnie sformatowany. Należy wybrać zbiór prostych zasad, które rządzą formatowaniem kodu, a następnie w konsekwentny sposób je stosować. Jeżeli pracujemy w zespole, to wszyscy jego członkowie powinni przyjąć zbiór zasad formatowania i stosować się do nich. Pomocne są narzędzia automatyczne, które stosują za nas te zasady formatowania.

Przeznaczenie formatowania

Na początku chciałbym coś ustalić. Formatowanie kodu *jest ważne*. Jest zbyt ważne, aby je ignorować, i zbyt ważne, aby traktować je dogmatycznie. Formatowanie kodu ma zapewnić komunikację, a dobra komunikacja jest pierwszą zasadą biznesu zawodowego programisty.

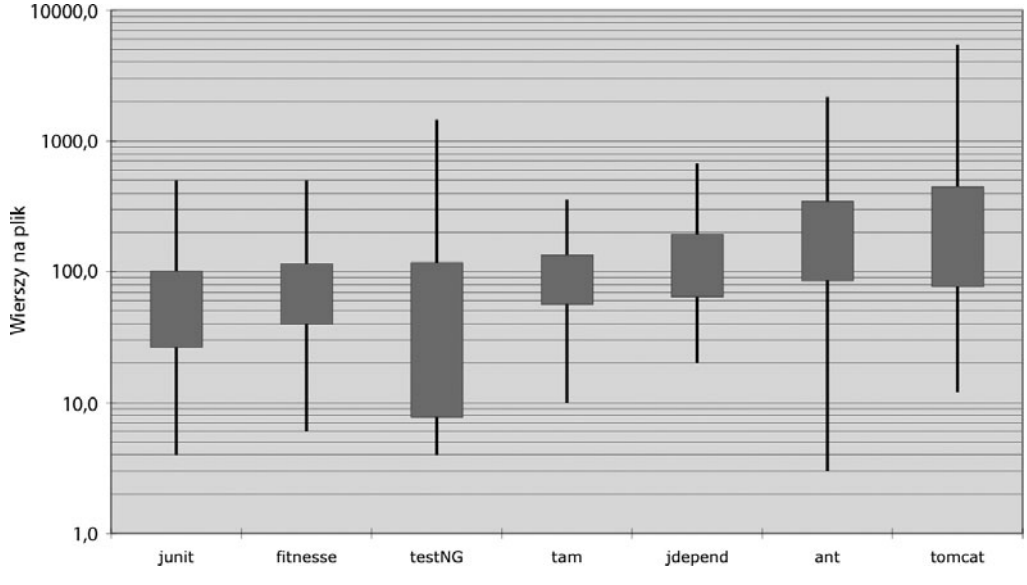
Być może ktoś stwierdzi, że pierwszą zasadą biznesu zawodowego programisty jest „zapewnienie działania". Mam nadzieję jednak, że już teraz książka ta dostatecznie skompromitowała to założenie. Funkcje, które tworzymy dziś, z dużym prawdopodobieństwem mogą się zmienić w następnej wersji, ale czytelność naszego kodu będzie miała zbawienny wpływ na wszystkie zmiany, jakie zostaną kiedykolwiek wykonane. Odpowiedni styl kodowania i czytelność kodu zapewniają łatwość utrzymania i rozszerzania kodu. Nasz styl i dyscyplina przetrwają, nawet jeżeli nasz kod już nie.

Jakie są więc zasady formatowania, które zapewniają uporządkowanie i czytelność kodu?

Formatowanie pionowe

Zaczniemy od rozmiaru pionowego. Jak duży powinien być kod źródłowy? W języku Java rozmiar pliku jest ściśle związany z wielkością klasy. Wielkość klas przedstawialiśmy przy okazji dyskusji o klasach. Teraz rozważmy tylko wielkość plików.

Jak duże są zwykle pliki źródłowe Java? Okazuje się, że różnice w zakresie wielkości i stylu kodu są znaczne (rys. 5.1).



RYSUNEK 5.1. Rozkład wielkości plików w skali logarytmicznej (wysokość prostokąta = sigma)

Przeanalizowano siedem różnych projektów — Junit, FitNesse, testNG, Time and Money, JDepend, Ant oraz Tomcat. Linie przechodzące przez prostokąty pokazują minimalną i maksymalną wielkość pliku w każdym projekcie. Prostokąt pokazuje średnio jedną trzecią (jedno standardowe odchylenie[1]) liczby plików. Środek prostokąta oznacza średnią. Tak więc średnia wielkość pliku w projekcie FitNesse wynosi około 65 wierszy, a około jedna trzecia plików ma od 40 do 100 i więcej wierszy. Największy plik w FitNesse ma około 400 wierszy, a najmniejszy 6 wierszy. Należy zwrócić uwagę, że jest to skala logarytmiczna, więc mała różnica w położeniu pionowym stanowi bardzo dużą różnicę w wielkości pliku.

JUnit, FitNesse oraz Time and Money są zbudowane ze względnie małych plików. Żaden z nich nie zawiera plików powyżej 500 wierszy, a większość plików ma poniżej 200 wierszy. Tomcat i Ant mają z kolei część plików o długości kilku tysięcy wierszy, a około połowa ma ponad 200 wierszy.

Co to oznacza? Wydaje się, że możliwe jest zbudowanie znaczącego systemu (FitNesse ma niemal 50 000 wierszy) z plików mających przeciętnie 200 wierszy, a nieprzekraczających 500 wierszy. Choć nie musi to być sztywna reguła, to jednak powinna być bardzo pożądana. Małe pliki są zwykle łatwiejsze do zrozumienia niż duże.

Metafora gazety

Pomyślmy o dobrze napisanym artykule w gazecie. Czytamy go od góry do dołu. Na górze oczekujemy nagłówka, który informuje nas o temacie danego artykułu i pozwala zdecydować, czy chcemy przeczytać całość. Pierwszy akapit przedstawia zarys artykułu, nie zawiera szczegółów. Gdy kontynuujemy lekturę w dół kolumny, liczba szczegółów zwiększa się — są to daty, nazwiska, cytaty i twierdzenia.

Chcemy, aby plik źródłowy był podobny do takiego artykułu w gazecie. Nazwa ma być prosta i sugestywna. Powinna wystarczyć nam do stwierdzenia, czy jesteśmy we właściwym module, czy nie. Górne partie pliku źródłowego mają zawierać algorytmy i koncepcje najwyższego poziomu. Liczba szczegółów powinna zwiększać się wraz ze schodzeniem w dół pliku, aż do funkcji oraz szczegółów na najniższym poziomie.

Gazeta składa się z wielu artykułów — w większości bardzo małych. Niektóre są nieco większe. Niewiele artykułów zawiera tyle tekstu, aby wypełnić całą stronę. Powoduje to, że gazety są *użyteczne*. Jeżeli gazeta przedstawiałaby jedną długą historię, zawierającą niezorganizowane konglomeraty faktów, dat i nazwisk, po prostu nie dałoby się jej czytać.

Pionowe odstępy pomiędzy segmentami kodu

Niemal każdy kod czyta się od lewej do prawej i od góry do dołu. Każdy wiersz reprezentuje wyrażenie lub klauzulę, a każda grupa wierszy reprezentuje kompletną myśl. Myśli te powinny być oddzielone od siebie pustymi wierszami.

---

[1] Prostokąt pokazuje wielkość sigma/2 powyżej i poniżej średniej. Tak, wiem, że rozkład wielkości pliku nie jest normalny, więc odchylenie standardowe nie jest matematycznie dokładne. Jednak nie chodzi tutaj o precyzję — chcę przedstawić trend.

Jako przykład weźmy kod z listingu 5.1. Znajdują się w nim puste wiersze oddzielające deklarację pakietu, importy i każdą z funkcji. Ta bardzo prosta zasada daje zbawienny efekt dla wizualnego układu kodu. Każdy pusty wiersz jest graficzną wskazówką identyfikującą nową, osobną koncepcję. Gdy przewijamy listing w dół, oko zatrzymuje się na pierwszym wierszu znajdującym się po pustym.

LISTING 5.1. BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Usunięcie tych pustych wierszy, tak jak na listingu 5.2, ma fatalny wpływ na czytelność kodu.

LISTING 5.2. BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Efekt ten jest nawet bardziej zauważalny, gdy spojrzymy na te bloki kodu z większej odległości. W pierwszym przykładzie poszczególne grupy wierszy będą dostrzegalne, natomiast drugi przykład będzie wyglądał na wymieszany. Jedyną różnicą pomiędzy tymi dwoma listingami jest zastosowanie kilku odstępów w pionie.

Gęstość pionowa

Jeżeli odstępy pozwalają na rozdzielenie koncepcji, to pionowe zagęszczenie pozwala wskazać związki. Dlatego wiersze kodu mające ze sobą ścisłe związki powinny być zagęszczone w pionie. Zwróćmy uwagę, jak bezużyteczne komentarze z listingu 5.3 łamią bliskie skojarzenie dwóch zmiennych instancyjnych.

LISTING 5.3.

```
public class ReporterConfig {

  /**
   * Nazwa klasy nasłuchu raportu.
   */
private String m_className;

  /**
   * Właściwości nasłuchu raportu.
   */
private List<Property> m_properties = new ArrayList<Property>();

public void addProperty(Property property) {
  m_properties.add(property);
}
```

Listing 5.4 jest znacznie łatwiejszy do czytania. Można go objąć wzrokiem, przynajmniej ja tak to odczuwam. Można na niego spojrzeć i stwierdzić, że jest to klasa z dwoma zmiennymi i metodami, bez potrzeby przemieszczania wzroku po kodzie. Aby osiągnąć ten sam poziom zrozumienia w przypadku wcześniejszego listingu, nie wystarczy jedno spojrzenie na kod.

LISTING 5.4.

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
}
```

Odległość pionowa

Czy kiedykolwiek szukałeś końca klasy, skakałeś z jednej funkcji do następnej, przewijałeś plik źródłowy w górę i w dół, próbując określić, jak są powiązane funkcje i jak działają, aby następnie znów zagubić się w labiryncie kodu? Czy kiedykolwiek śledziłeś łańcuch dziedziczenia dla definicji zmiennej lub funkcji? Jest to frustrujące, ponieważ chcemy zrozumieć, *co* robi system, a tracimy czas i energię na wyszukanie i zapamiętanie, *gdzie* znajdują się jego składniki.

Koncepcje, które są ze sobą ściśle związane, powinny znajdować się w niewielkiej odległości od siebie [G10]. Jasne jest, że zasada ta nie może być stosowana w przypadku koncepcji znajdujących się w osobnych plikach. Jednak ściśle powiązane koncepcje nie powinny być umieszczane w różnych plikach, o ile nie ma ku temu specjalnych przesłanek. Jest to jeden z powodów, dla których należy unikać zmiennych chronionych.

W przypadku tych koncepcji, które są ze sobą tak ściśle powiązane, że należą do tego samego pliku źródłowego, ich odległość od siebie jest miarą tego, jak dana funkcja jest ważna dla zrozumienia drugiej. Nie powinniśmy zmuszać czytelników do skakania po pliku źródłowym i klasach.

Deklaracje zmiennych. Zmienne powinny być zadeklarowane tak blisko miejsca ich użycia, jak tylko to możliwe. Ponieważ nasze zmienne są bardzo krótkie, zmienne lokalne powinny znajdować się na początku każdej z funkcji, tak jak w tych długich funkcjach z Junit 4.3.1.

```
private static void readPreferences() {
  InputStream is= null;
    try {
      is= new FileInputStream(getPreferencesFile());
      setPreferences(new Properties(getPreferences()));
      getPreferences().load(is);
    } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

Zmienne sterujące pętli powinny być deklarowane wewnątrz instrukcji pętli, tak jak w małych funkcjach z tego samego źródła.

```
public int countTestCases() {
  int count= 0;
  for (Test each : tests)
    count += each.countTestCases();
  return count;
}
```

W rzadkich przypadkach zmienne mogą być deklarowane na początku bloku lub w dłuższej funkcji, bezpośrednio przed pętlą. Można spotkać zmienne z poniższego fragmentu w środku bardzo długiej funkcji z TestNG.

```
...
for (XmlTest test : m_suite.getTests()) {
      TestRunner tr = m_runnerFactory.newTestRunner(this, test);
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);

      invoker = tr.getInvoker();

      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
            beforeSuiteMethods.put(m.getMethod(), m);
      }

      for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
            afterSuiteMethods.put(m.getMethod(), m);
      }
   }
...
```

Zmienne instancyjne z kolei powinny być deklarowane na początku klasy. Nie należy zwiększać odległości pionowej tych zmiennych, ponieważ w dobrze zaprojektowanej klasie są używane przez wiele metod klasy, o ile nie przez wszystkie.

Od dawna toczą się debaty na temat tego, gdzie powinny znajdować się zmienne instancyjne. W języku C++ powszechnie stosowaliśmy tak zwaną *regułę nożyczek*, zgodnie z którą umieszczaliśmy wszystkie zmienne instancyjne na końcu. Z kolei w języku Java powszechnie stosowaną konwencją jest umieszczanie wszystkich takich zmiennych na początku klasy. Nie widzę powodów, dla których mielibyśmy stosować się do którejkolwiek z tych konwencji. Najważniejsze dla zmiennych instancyjnych jest to, aby były zadeklarowane w jednym znanym miejscu. Każdy powinien wiedzieć, gdzie zajrzeć, aby zobaczyć deklaracje.

Jako przykład weźmy dziwny przypadek klasy `TestSuite` z JUnit 4.3.1. Musiałem znacznie rozrzedzić tę klasę, aby zorientować się, jak ona działa. Jeżeli spojrzymy na środkową część listingu, zauważymy zadeklarowane tam dwie zmienne instancyjne. Trudno byłoby schować je w lepszym miejscu. Osoba czytająca ten kod mogłaby przypadkowo ominąć te deklaracje (tak jak ja).

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
     ...
  }

  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
  ...
  }

  public static Test warning(final String message) {
  ...
  }

  private static String exceptionToString(Throwable t) {
  ...
  }

  private String fName;

  private Vector<Test> fTests= new Vector<Test>(10);

  public TestSuite() {
  }

  public TestSuite(final Class<? extends TestCase> theClass) {
  ...
  }

  public TestSuite(Class<? extends TestCase> theClass, String name) {
  ...
  }
... ... ... ... ...
}
```

Funkcje zależne. Jeżeli jedna funkcja wywołuje inną, powinny być one położone blisko siebie, a funkcja wywołująca powinna być umieszczona powyżej wywoływanej, o ile jest to możliwe. Pozwala to osiągnąć naturalny przebieg programu. Jeżeli konwencja ta jest stosowana konsekwentnie, czytelnik będzie mógł zaufać nam, że definicja funkcji znajduje się zaraz po jej wywołaniu. Jako przykład weźmy fragment z FitNesse z listingu 5.5. Warto zwrócić uwagę, że funkcje umieszczone na górze wywołują te znajdujące się poniżej, które z kolei wywołują położone jeszcze niżej. Powoduje to, że łatwiej można znaleźć wywoływane funkcje, i znacznie poprawia czytelność całego modułu.

LISTING 5.5. WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
  ...
```

Przy okazji mamy w tym fragmencie przykład rozmieszczania stałych na odpowiednim poziomie [G35]. Stała `FrontPage` mogłaby być umieszczona w funkcji `getPageNameOrDefault`, ale spowodowałoby to ukrycie znanej i oczekiwanej stałej w funkcji znajdującej się na zbyt niskim poziomie. Lepiej było przekazywać tę stałą w dół, z miejsca, gdzie jej definicja ma sens, do miejsca, w którym jest wykorzystywana.

Koligacja koncepcyjna. Niektóre fragmenty kodu chcemy umieszczać niedaleko innych fragmentów. Mają one określoną koligację koncepcyjną. Im silniejsza ta koligacja, tym mniejsza powinna być odległość pomiędzy tymi fragmentami.

Jak można się przekonać, koligacja ta może bazować na bezpośredniej zależności, takiej jak wywoływanie jednej funkcji przez inną lub użycie zmiennej w funkcji. Istnieją również inne możliwości koligacji. Koligacja może wynikać z tego, że funkcje wykonują podobne operacje. Weźmy pod uwagę fragment kodu zaczerpnięty z Junit 4.3.1:

```
public class Assert {
  static public void assertTrue(String message,
  ↪boolean condition) {
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

  static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
...
```

Funkcje te mają silną koligację koncepcyjną, ponieważ współużytkują ten sam schemat nazewnictwa i wykonują odmiany tego samego zadania. Fakt, że wywołują się wzajemnie, jest mniej ważny. Nawet jeżeli nie robiłyby tego, nadal chcielibyśmy trzymać je razem.

Uporządkowanie pionowe

Zwykle chcemy, aby w funkcji wywołania zależności były kierowane w dół. Inaczej mówiąc, wywoływana funkcja powinna znajdować się poniżej funkcji wywołującej[2]. Daje to efekt przepływu sterowania w dół kodu źródłowego modułu, od wysokiego poziomu do niskiego.

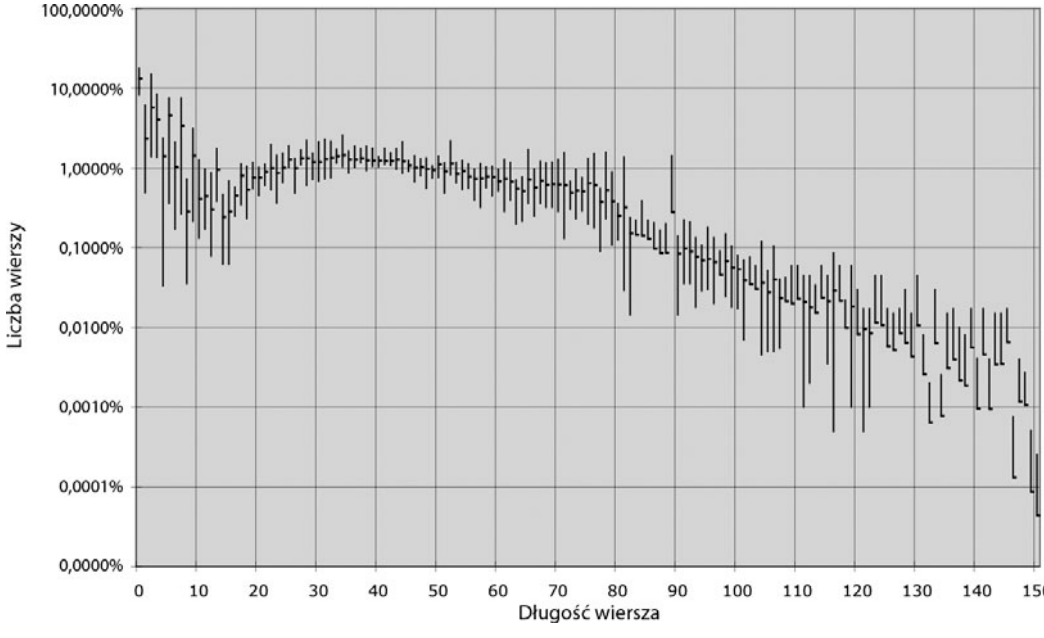
W artykule prasowym oczekujemy, że najważniejsze informacje znajdą się na początku, a następnie zostaną opisane z najmniejszą liczbą przeszkadzających detali. Oczekujemy, że najdrobniejsze szczegóły będą przedstawione na końcu. Podobnie w przypadku kodu — pozwala to nam zapoznać się z plikami źródłowymi i uzyskiwać ich obraz na podstawie kilku pierwszych funkcji, bez zagłębiania się w szczegóły. W taki sposób jest zorganizowany kod z listingu 5.5. Jeszcze lepszymi przykładami są listingi 15.5 i 3.7.

---

[2] Jest to odwrotna sytuacja w stosunku do języków takich jak Pascal, C i C++, w których funkcje musiały być zdefiniowane lub co najmniej zadeklarowane *przed* ich użyciem.

Formatowanie poziome

Jak długie powinny być wiersze? Aby odpowiedzieć na to pytanie, sprawdzimy długość wierszy w typowych programach. Ponownie przeanalizujemy siedem różnych projektów. Na rysunku 5.2 przedstawiony jest rozkład długości wierszy wszystkich siedmiu projektów. Regularność jest imponująca, szczególnie około 45 znaków. W rzeczywistości każda wartość od 20 do 60 reprezentuje około 1 procent całkowitej liczby wierszy. To jest 40 procent! Kolejne 30 procent to wiersze krótsze niż 10 znaków. Należy pamiętać, że jest to skala logarytmiczna, więc liniowy spadek długości powyżej 80 znaków jest naprawdę bardzo znaczący. Jasne jest, że programiści preferują krótkie wiersze.



RYSUNEK 5.2. Rozkład długości wierszy kodu Java

A zatem powinniśmy tworzyć krótkie wiersze. Stare ograniczenie Holleritha do 80 znaków jest zbyt restrykcyjne i osobiście nie mam nic przeciwko wierszom mającym 100, a nawet 120 znaków. Jednak wartości powyżej tej granicy są zbyt duże.

Sam piszę programy tak, aby nigdy nie trzeba było przewijać ekranu edytora w prawo. Jednak obecnie monitory są bardzo szerokie, a młodsi programiści mogą tak zmniejszyć czcionkę, że uzyskują wiersze o długości 200 znaków. Nie jest to dobra praktyka. Ja staram się nie przekraczać długości 120 znaków.

Poziome odstępy i gęstość

Odstępy poziome służą do kojarzenia elementów ściśle powiązanych i rozłączania elementów, które są ze sobą luźniej związane. Weźmy pod uwagę następującą funkcję:

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Operatory przypisania otoczyłem odstępami w celu ich wyróżnienia. Instrukcje przypisania mają dwa osobne główne elementy: lewą i prawą stronę. Odstępy powodują, że rozdzielenie to jest oczywiste.

Z drugiej strony, nie dodaję odstępów pomiędzy nazwami funkcji a otwierającymi nawiasami. Funkcje i argumenty są ze sobą bardzo ściśle związane. Rozdzielenie ich spowodowałoby, że wyglądałyby na rozłączone, a nie złączone. Rozdzielam argumenty wywołania funkcji, aby zaznaczyć przecinek i pokazać, że argumenty są osobne.

Innym zastosowaniem odstępów jest zaznaczenie kolejności wykonywania operatorów.

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

Warto zwrócić uwagę, jak łatwo czyta się te wyrażenia. Wykładniki nie są oddzielone odstępami, ponieważ mają wysoki priorytet. Składniki są rozdzielone, ponieważ operacje dodawania i odejmowania mają niski priorytet.

Niestety, większość narzędzi formatujących kod jest ślepa na priorytety operatorów i stosuje wszędzie takie same odstępy. Tak więc subtelne odstępy, takie jak tu przedstawione, zwykle są tracone po automatycznym sformatowaniu kodu.

Rozmieszczenie poziome

Gdy byłem programistą asemblera[3], stosowałem rozmieszczenie poziome w celu wyróżniania określonych struktur. Gdy zacząłem kodować w C, C++ i w końcu w Javie, nadal starałem się wyrównywać wszystkie nazwy zmiennych w zbiorze deklaracji lub wszystkie r-wartości w zbiorze instrukcji przypisania. Kod taki może wyglądać następująco:

[3] Czy ja żartuję? Nadal jestem programistą asemblera. Można odsunąć chłopca od zabawek, ale trudno mu je zabrać!

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket          socket;
  private   InputStream     input;
  private   OutputStream    output;
  private   Request         request;
  private   Response        response;
  private   FitNesseContext context;
  protected long            requestParsingTimeLimit;
  private   long            requestProgress;
  private   long            requestParsingDeadline;
  private   boolean         hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
}
```

Zauważyłem jednak, że tego typu wyrównywanie nie jest użyteczne. Wydaje się, że wyrównanie takie powoduje skupienie uwagi na niewłaściwych elementach i odciąga wzrok od właściwych. Na przykład w przedstawionej powyżej liście deklaracji mamy pokusę czytania listy zmiennych bez zwracania uwagi na ich typy. Podobnie w instrukcjach przypisania zwykle patrzymy na listę r-wartości, bez zwracania uwagi na operator przypisania. Co gorsza, automatyczne narzędzia formatujące zwykle eliminują tego rodzaju wyrównanie.

Z tego powodu w końcu przestałem stosować ten typ formatowania. Obecnie preferuję niewyrównane deklaracje i przypisania, tak jak w poniższym przykładzie, ponieważ wskazują one na ważną niedoskonałość. Jeżeli mam długą listę, która powinna być wyrównana, występuje *problem z szerokością listy*, a nie z brakiem wyrównania. Długość listy deklaracji w `FitNesseExpediter`, znajdującej się poniżej, sugeruje konieczność podziału klasy.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;

  public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
  {
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Wcięcia

Plik źródłowy stanowi pewnego rodzaju hierarchię, a nie szablon. Znajdują się w nim informacje odnoszące się do pliku jako całości, do poszczególnych klas w pliku, do metod w klasach, do bloków w metodach oraz, rekurencyjnie, do bloków w blokach. Każdy poziom hierarchii jest zakresem, w którym są zadeklarowane nazwy oraz w którym są interpretowane deklaracje i instrukcje wykonywalne.

Aby hierarchia ta była widoczna, wcinamy wiersze kodu źródłowego odpowiednio do pozycji w hierarchii. Instrukcje na poziomie pliku, takie jak większość deklaracji klas, nie są wcinane. Metody w klasach są wcinane jeden poziom w prawą stronę klasy. Implementacje tych metod znajdują się jeden poziom w prawo w stosunku do deklaracji metody. Implementacje bloków znajdują się jeden poziom w prawo w stosunku do bloku, w którym się znajdują, i tak dalej.

Programiści polegają w znacznym stopniu na tym schemacie wcięć. Wyrównują oni wizualnie wiersze z lewej strony, aby sprawdzić, w którym zakresie się znajdują. Pozwala to szybko przeskakiwać zakresy, takie jak implementacje instrukcji `if` lub `while`, które nie odnoszą się do bieżącej sytuacji. Od lewej strony programiści szukają deklaracji metod, nowych zmiennych, a nawet nowych klas. Bez wcięć programy byłyby niemal nieczytelne dla ludzi.

Spójrzmy na poniższe programy, które są pod względem składniowym i semantycznym identyczne:

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
-----


public class FitNesseServer implements SocketServer {
  private FitNesseContext context;

  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Nasze oko może szybko określić strukturę wcięć w pliku. Można niemal natychmiast odszukać zmienne, konstruktory, akcesory i metody. Tylko kilka sekund zajmuje określenie, że jest to jakiś interfejs do gniazda z obsługą czasu oczekiwania. Wersja bez wcięć jest jednak niemal nieczytelna bez intensywnego studiowania.

Łamanie wcięć. Czasami kuszące jest łamanie zasad wcięć w przypadku krótkich instrukcji `if`, krótkich pętli `while` lub krótkich funkcji. Wszędzie tam, gdzie uległem tej pokusie, niemal zawsze wracałem i dodawałem właściwe wcięcia. Dlatego należy unikać łączenia zakresów w jeden wiersz, tak jak w poniższym przykładzie:

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

Polecam rozwinięcie kodu i wcięcie zakresów, tak jak poniżej:

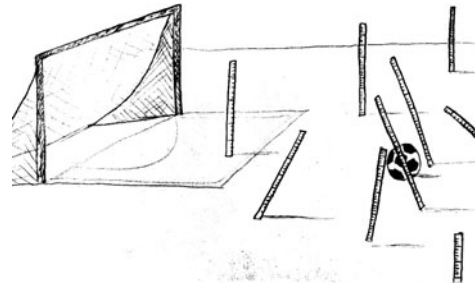
```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```

Puste zakresy

Czasami treść instrukcji `while` lub `for` nie zawiera kodu, tak jak jest to pokazane poniżej. Nie lubię tego typu struktur i staram się ich unikać. Gdy nie mogę ich uniknąć, upewniam się, że pusta treść jest prawidłowo wcięta i otoczona klamrami. Nie zliczę, ile razy dałem się oszukać przez średnik siedzący cichutko na końcu pętli `while` w tym samym wierszu. Jeżeli nie zapewnimy, że średnik ten będzie *widoczny*, przez zastosowanie wcięcia w jego własnym wierszu, trudno będzie go zauważyć.

```
while (dis.read(buf, 0, readBufferSize) != -1)
  ;
```

Zasady zespołowe



Tytuł tego podrozdziału jest grą słów. Każdy programista ma własne ulubione zasady formatowania, ale jeżeli pracuje w zespole, zasady ustala zespół.

Zespół programistów powinien ustalić jeden styl formatowania, a następnie każdy członek zespołu powinien stosować ten styl. Oczekujemy, że program będzie miał jednorodny styl. Nie chcemy, aby wyglądał, jakby był napisany przez bandę niezgadzających się ze sobą indywiduów.

Gdy w roku 2002 zacząłem pracę nad projektem FitNesse, usiadłem z zespołem w celu wypracowania stylu kodowania. Zajęło to około 10 minut. Zdecydowaliśmy, gdzie będziemy umieszczać klamry, jaka powinna być wielkość wcięcia, jak będziemy nazywać klasy, zmienne, metody i tak dalej. Następnie wpisaliśmy te zasady w formater kodu naszego IDE i od tego momentu je stosowaliśmy. Nie były to zasady, które preferuję; były to zasady, o których zdecydował zespół. Jako członek zespołu stosowałem je przy pisaniu kodu w projekcie FitNesse.

Należy pamiętać, że dobre oprogramowanie składa się ze zbioru dokumentów, które dobrze się czyta. Muszą mieć one spójny i gładki styl. Muszą budzić zaufanie czytelnika, który uzna, że formatowanie, jakie zobaczy w jednym pliku źródłowym, będzie znaczyło to samo w innym. Ostatnią rzeczą, jaką chcemy zrobić, jest zwiększenie skomplikowania kodu przez jego pisanie przy użyciu kilku różnych odrębnych stylów.

Zasady formatowania wujka Boba

Zasady formatowania stosowane przeze mnie są bardzo proste i zostały przedstawione w kodzie na listingu 5.6. Można go uznać za przykład tego, jak kod tworzy najlepszy dokument standardu kodowania.

LISTING 5.6. CodeAnalyzer.java

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;

  public CodeAnalyzer() {
    lineWidthHistogram = new LineWidthHistogram();
  }

  public static List<File> findJavaFiles(File parentDirectory) {
    List<File> files = new ArrayList<File>();
    findJavaFiles(parentDirectory, files);
    return files;
  }

  private static void findJavaFiles(File parentDirectory, List<File> files) {
    for (File file : parentDirectory.listFiles()) {
      if (file.getName().endsWith(".java"))
        files.add(file);
      else if (file.isDirectory())
        findJavaFiles(file, files);
    }
  }

  public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader(javaFile));
```

Więcej na: www.ebook4all.pl

```
    String line;
    while ((line = br.readLine()) != null)
      measureLine(line);
  }

  private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
  }

  private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
      maxLineWidth = lineSize;
      widestLineNumber = lineCount;
    }
  }

  public int getLineCount() {
    return lineCount;
  }

  public int getMaxLineWidth() {
    return maxLineWidth;
  }

  public int getWidestLineNumber() {
    return widestLineNumber;
  }

  public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
  }

  public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
  }

  public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
      cumulativeLineCount += lineCountForWidth(width);
      if (cumulativeLineCount > lineCount/2)
        return width;
    }
    throw new Error("Nie można tu wejść");
  }

  private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
  }

  private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
  }
}
```

Obiekty i struktury danych

STNIEJE POWÓD, DLA KTÓREGO korzystamy ze zmiennych prywatnych. Nie chcemy, aby ktokolwiek na nich polegał. Chcemy zachować sobie swobodę zmian ich typu lub implementacji. Dlaczego więc tak wielu programistów automatycznie dodaje gettery i settery do swoich obiektów, udostępniając zmienne prywatne tak, jakby były one publiczne?

Abstrakcja danych

Zwróćmy uwagę na różnice pomiędzy listingiem 6.1 a listingiem 6.2. Oba reprezentują dane punktu w układzie kartezjańskim. A jednak w pierwszym przypadku implementacja jest udostępniona, a w drugim całkowicie ukryta.

LISTING 6.1. Punkt konkretny

```
public class Point {
  public double x;
  public double y;
}
```

LISTING 6.2. Punkt abstrakcyjny

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Najlepsze w kodzie z listingu 6.2 jest to, że nie ma sposobu na stwierdzenie, czy implementacja korzysta ze współrzędnych prostokątnych, czy kątowych. Być może całkowicie innych? A jednak interfejs niezawodnie reprezentuje strukturę danych.

Co lepsze, reprezentuje więcej niż tylko strukturę danych. Metody pozwalają na wymuszenie zasad dostępu. Można niezależnie odczytywać poszczególne współrzędne, ale ustawianie współrzędnych musi być wykonywane jako atomowa (tzn. pojedyncza) operacja.

Z drugiej strony, na listingu 6.1 jest bardzo wyraziście zaimplementowana klasa współrzędnych prostokątnych, która zmusza nas do niezależnego manipulowania tymi współrzędnymi. Powoduje to ujawnienie implementacji. W rzeczywistości implementacja ta byłaby również ujawniona, jeżeli zmienne byłyby prywatne i używalibyśmy setterów i getterów poszczególnych zmiennych.

Ukrywanie implementacji nie sprowadza się do dodawania warstwy funkcji nad zmiennymi. Ukrywanie implementacji polega na tworzeniu abstrakcji! Klasa nie powinna po prostu przepychać zmiennych przez gettery i settery. Zamiast tego powinna udostępniać interfejs pozwalający użytkownikom na manipulowanie *istotą* danych bez konieczności znajomości jej implementacji.

Spójrzmy na listingi 6.3 i 6.4. W pierwszym użyte są konkretne terminy informujące o poziomie paliwa w pojeździe, natomiast w drugim użyta jest abstrakcja poziomu procentowego. W konkretnym przypadku możemy być niemal pewni, że są to po prostu akcesory zmiennych. W przypadku abstrakcyjnym nie mamy pojęcia o formie danych.

LISTING 6.3. Pojazd konkretny

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

LISTING 6.4. Pojazd abstrakcyjny

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

W obu przedstawionych przypadkach zalecane jest stosowanie drugiego z przedstawionych kodów. Nie chcemy udostępniać szczegółów naszych danych. Zamiast tego powinniśmy opisywać dane abstrakcyjnymi terminami. Nie można tego osiągnąć przez proste użycie interfejsów oraz getterów i setterów. Konieczna jest dogłębna analiza w celu określenia najlepszego sposobu reprezentowania danych znajdujących się w obiektach. Najgorszą opcją jest ślepe dodanie getterów i setterów.

Antysymetria danych i obiektów

Poniższe dwa przykłady pokazują różnice pomiędzy obiektami i strukturami danych. Obiekty ukrywają dane, tworząc abstrakcje, i udostępniają funkcje operujące na tych danych. Struktury danych udostępniają ich dane i nie mają znaczących funkcji. Przeczytajmy to jeszcze raz. Zwróćmy uwagę na komplementarną naturę tych dwóch definicji. Są one właściwie swoimi przeciwieństwami. Różnica ta może wydawać się nieznaczna, ale ma daleko idące implikacje.

Jako przykład weźmy proceduralny kod kształtu z listingu 6.5. Klasa `Geometry` operuje na trzech klasach kształtu. Klasy kształtów są prostymi strukturami danych bez własnych operacji. Wszystkie operacje są zdefiniowane w klasie `Geometry`.

LISTING 6.5. Kształt proceduralny

```
public class Square {
  public Point topLeft;
  public double side;
  }

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}

public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Czytelnicy, którzy preferują programowanie zorientowane obiektowo, mogą zmarszczyć brwi i narzekać, że jest to kod proceduralny — i będą mieli rację. Jednak to szyderstwo może być nie na miejscu. Zwróćmy uwagę, co się stanie, jeżeli dodamy funkcję `perimeter()` do klasy `Geometry`. Klasy kształtów pozostaną bez zmian! Wszystkie inne klasy korzystające z klas kształtów również się nie zmienią! Z drugiej strony, jeżeli dodamy nowy kształt, musimy zmienić wszystkie operujące na nim funkcje z `Geometry`. Przeczytajmy to ponownie. Trzeba zwrócić uwagę, że te dwa warunki są zupełnie przeciwstawne.

Weźmy teraz przykład obiektowy z listingu 6.6. Metoda `area()` jest tu polimorficzna. Nie jest potrzebna klasa `Geometry`. Jeżeli więc dodamy nowy kształt, żadna z istniejących *funkcji* nie ulegnie zmianie, ale jeżeli dodamy nową funkcję, wszystkie *kształty* będą musiały być zmienione[1]!

LISTING 6.6. Kształt polimorficzny

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}

public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Ponownie widzimy komplementarną naturę tych dwóch definicji — są one swoimi przeciwieństwami! Pokazują podstawową różnicę pomiędzy obiektami i strukturami danych:

> *Kod proceduralny (kod korzystający ze struktur danych) ułatwia dodawanie nowych funkcji bez zmiany istniejących struktur danych. Z kolei kod obiektowy ułatwia dodawanie nowych klas bez zmiany istniejących funkcji.*

[1] Istnieją sposoby obejścia tego problemu, dobrze znane doświadczonym projektantom obiektowym. Przykładem może być wzorzec `Visitor` lub podwójne rozsyłanie. Jednak techniki te są obciążone własnymi kosztami i zwykle dają w wyniku strukturę znaną z podejścia proceduralnego.

Prawdziwe jest również stwierdzenie odwrotne:

> *Kod proceduralny utrudnia dodawanie nowych struktur danych, ponieważ muszą zostać zmienione wszystkie funkcje. Kod obiektowy utrudnia dodawanie nowych funkcji, ponieważ muszą zostać zmienione wszystkie klasy.*

Tak więc operacje, które są trudne do wykonania w sposób obiektowy, można łatwo zrealizować przy użyciu procedur, a operacje trudne do realizacji za pomocą procedur można łatwo wykonać za pomocą obiektów!

A co zrobić w sytuacji, kiedy w złożonym systemie chcemy dodać nowy typ danych zamiast nowych funkcji? Dla takich przypadków najwłaściwsze jest zastosowanie technik obiektowych. Z drugiej strony, zdarza się również, że chcemy dodać nowe funkcje. W takich przypadkach najwłaściwszy wydaje się kod proceduralny i struktury danych.

Doświadczeni programiści wiedzą, że twierdzenie, iż wszystko jest obiektem, *to mit*. Czasami faktycznie *warto* zastosować proste struktury danych z operującymi na nich procedurami.

Prawo Demeter

Dobrze znana zasada, nazywana *prawem Demeter*[2], mówi, że moduł powinien nie wiedzieć nic o wnętrzu *obiektów*, którymi manipuluje. Jak przedstawiliśmy w poprzednim punkcie, obiekty ukrywają swoje dane i udostępniają operacje. Oznacza to, że obiekt nie powinien udostępniać swojej struktury wewnętrznej przy użyciu akcesorów, ponieważ powoduje to udostępnienie, a nie ukrycie wewnętrznej struktury.

Dokładniej rzecz ujmując, prawo Demeter głosi, że metoda `f` klasy `C` powinna wywoływać tylko metody z:

- `C`,
- obiektu utworzonego przez `f`,
- obiektu przekazanego jako argument do `f`,
- obiektu umieszczonego w zmiennej instancyjnej klasy `C`.

Metoda *nie powinna* wywoływać metod z obiektów zwracanych przez jakąkolwiek z innych dozwolonych funkcji. Inaczej mówiąc, powinniśmy rozmawiać z przyjaciółmi, a nie z obcymi.

W poniższym kodzie[3] nie jest zachowane prawo Demeter (między innymi), ponieważ wywoływana jest metoda `getScratchDir()` z obiektu zwracanego przez `getOptions()`, a następnie wywoływana jest metoda `getAbsolutePatch()` na obiekcie zwracanym przez `getScratchDir()`.

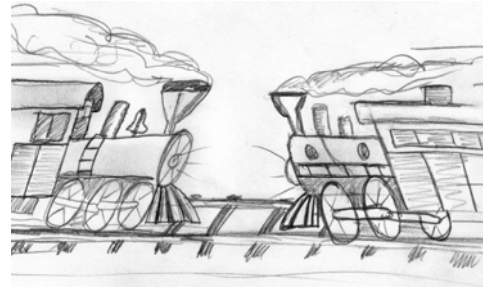
```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

---

[2] *http://en.wikipedia.org/wiki/Law_of_Demeter*

[3] Znalezionym gdzieś w bibliotekach Apache.

Wraki pociągów



Przedstawiony powyżej kod jest często nazywany wrakiem pociągu, ponieważ wygląda jak kilka wagonów kolejowych po zderzeniu. Tego rodzaju łańcuchy wywołań są uznawane za oznakę słabego stylu programowania i należy ich unikać [G36]. Najlepiej podzielić je w następujący sposób:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Czy te dwa fragmenty kodu naruszają prawo Demeter? Oczywiście wiemy, iż w tym module obiekt `ctxt` zawiera opcje, które zawierają katalog pośredni, który z kolei posiada ścieżkę bezwzględną. To sporo wiedzy jak na jedną funkcję. Wywołująca funkcja wie, w jaki sposób nawigować pomiędzy wieloma różnymi obiektami.

To, czy naruszone jest prawo Demeter, zależy od tego, czy `ctxt`, `Options` i `ScratchDir` są obiektami, czy strukturami danych. Jeżeli są obiektami, to ich wewnętrzna struktura powinna być ukryta, a nie ujawniona, więc wiedza na temat jej budowy wewnętrznej jest jawnym naruszeniem prawa Demeter. Z drugiej strony, jeżeli `ctxt`, `Options` i `ScratchDir` są po prostu strukturami danych bez zdefiniowanych operacji, to w naturalny sposób ujawniają swoją strukturę wewnętrzną i prawo Demeter nie ma tu zastosowania.

Zastosowanie funkcji akcesorów zwykle utrudnia dostrzeżenie problemu. Jeżeli kod zostałby napisany tak jak poniżej, to prawdopodobnie nie pytalibyśmy o naruszenie prawa Demeter.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Problem ten byłby znacznie mniej kłopotliwy, jeżeli struktury danych po prostu miałyby zmienne publiczne i nie zawierały funkcji, natomiast obiekty miałyby zmienne prywatne i funkcje publiczne. Jednak istnieją biblioteki i standardy (na przykład *beans*), które wymagają, aby nawet proste struktury danych miały akcesory i mutatory.

Hybrydy

Przedstawione problemy często prowadzą do powstania struktur hybrydowych będących w połowie obiektami, a w połowie strukturami danych. Mają one funkcje wykonujące ważne operacje, ale mają również zmienne publiczne lub publiczne akcesory i mutatory, które oprócz tego, że realizują zamierzone operacje, udostępniają zmienne prywatne, co pozwala zewnętrznym funkcjom na użycie tych zmiennych w sposób, w jaki programy proceduralne używają struktur danych[4].

Takie hybrydy utrudniają dodawanie nowych funkcji, jak również utrudniają dodawanie nowych struktur danych. Mają najgorsze cechy z obu światów. Należy unikać ich tworzenia. Są one charakterystyczne dla tych projektów, w których autorzy nie są pewni — lub co gorsza, nie mają pojęcia — czy potrzebują ochrony funkcji, czy typów.

---

[4] Jest to czasami nazywane „zazdrością o funkcje" (ang. *Feature Envy*) [Refactoring].

Ukrywanie struktury

Załóżmy, że `ctxt`, `options` oraz `scratchDir` są obiektami ze zdefiniowanymi operacjami. W takim przypadku obiekty powinny ukrywać swoją wewnętrzną strukturę i nie powinniśmy mieć możliwości nawigowania po nich. W jaki sposób możemy więc otrzymać ścieżkę bezwzględną do katalogu pośredniego?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

lub

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

Pierwsza opcja może doprowadzić do eksplozji metod w obiekcie `ctxt`. W drugiej zakładamy, że `getScratchDirectoryOption()` zwraca strukturę danych, a nie obiekt. Żadna z tych opcji nie wygląda dobrze.

Jeżeli `ctxt` jest obiektem, powinniśmy powiedzieć mu, aby *coś zrobił*; nie powinniśmy pytać go o szczegóły wewnętrzne. Dlaczego więc chcemy otrzymać ścieżkę bezwzględną do katalogu pośredniego? Co mamy zamiar z nią zrobić? Możemy się dowiedzieć tego z następującego kodu (znajdującego się wiele wierszy dalej) z tego samego modułu:

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Wymieszanie różnych poziomów szczegółowości [G34][G6] powoduje problemy. Kropki, ukośniki i obiekty `File` nie powinny być tak beztrosko mieszane ze sobą oraz z otaczającym je kodem. Ignorując ten problem, widzimy w końcu, że przeznaczeniem ścieżki bezwzględnej do katalogu pośredniego było utworzenie pliku roboczego o podanej nazwie.

Czy obiekt `ctxt` może to zrobić za nas?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Wydaje się to rozsądną operacją, jaką obiekt może wykonać! Pozwala to ukryć szczegóły wewnętrzne obiektu `ctxt` i uniemożliwia funkcjom łamanie prawa Demeter przez nawigowanie pomiędzy obiektami, o których nie powinniśmy wiedzieć.

Obiekty transferu danych

Kwintesencją postaci struktur danych jest klasa ze zmiennymi publicznymi niezawierająca funkcji. Czasami jest to nazywane obiektem transferu danych, czyli DTO. DTO są bardzo przydatnymi strukturami, szczególnie w przypadku komunikowania się z bazami danych, analizowania komunikatów z gniazd sieciowych i tak dalej. Często stają się pierwszym elementem serii przekształceń, które konwertują surowe dane z bazy danych na obiekty w kodzie aplikacji.

Nieco częściej spotykaną jest postać *bean*, przedstawiona na listingu 6.7. Ma ona zmienne prywatne, na których operuje się za pomocą getterów i setterów. Ta quasi-hermetyzacja obiektów *bean* powoduje, że niektórzy puryści obiektowi czują się lepiej, ale nie ma ona żadnych dodatkowych zalet.

LISTING 6.7. address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

Active Record

Active Record to specjalna forma DTO. Są to struktury danych ze zmiennymi publicznymi (lub dostępnymi w stylu *bean*), ale dodatkowo mają metody nawigacyjne, takie jak `save` i `find`. Zwykle obiekty Active Record są bezpośrednim tłumaczeniem z tabel bazy danych lub innych źródeł.

Niestety, często zdarza się, że programiści traktują te struktury danych, jakby były pełnoprawnymi obiektami, i umieszczają w nich metody zasad biznesowych. Jest to dziwne, ponieważ powoduje powstanie hybrydy struktury danych i obiektu.

Rozwiązaniem jest oczywiście traktowanie Active Record jako struktury danych i tworzenie osobnych obiektów, które zawierają zasady biznesowe i ukrywają ich dane wewnętrzne (którymi będą prawdopodobnie obiekty Active Record).

Zakończenie

Obiekty udostępniają operacje i ukrywają dane. Ułatwia to dodawanie nowych rodzajów obiektów bez zmiany istniejących operacji, ale utrudnia dodawanie nowych operacji do istniejących obiektów. Struktury danych udostępniają dane i nie mają znaczących operacji. Ułatwia to dodawanie nowych operacji do istniejących struktur danych, ale utrudnia dodawanie nowych struktur danych do istniejących funkcji.

W każdym systemie potrzebujemy czasami elastyczności w dodawaniu nowych typów danych, więc w tych częściach systemu powinniśmy korzystać głównie z obiektów. W innych przypadkach oczekujemy elastyczności w dodawaniu nowych operacji, więc w tych częściach systemu powinniśmy korzystać głównie z typów danych i procedur. W programowaniu nie należy kierować się uprzedzeniami, ale wybierać takie rozwiązanie, które umożliwi najlepszą realizację zadań.

Bibliografia

[Refactoring]: Martin Fowler i inni, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999.

Obsługa błędów

Michael Feathers

ROZDZIAŁ POŚWIĘCONY OBSŁUDZE BŁĘDÓW w książce na temat czystego kodu może wydawać się nieporozumieniem. A jednak musimy zająć się tym tematem. Obsługa błędów jest po prostu częścią programowania. Dane wejściowe mogą być nieprawidłowe, a urządzenia mogą ulec uszkodzeniu. Mówiąc krótko, wszystko może pójść źle, a programista jest odpowiedzialny za takie skonstruowanie kodu, aby wykonywał to, co powinien.

Połączenie tematu obsługi błędów z zagadnieniami czystego kodu powinno być oczywiste. Wiele zbiorów kodu jest całkowicie zdominowanych przez obsługę błędów. Mówiąc „zdominowanych", nie mam na myśli tego, że kod ten realizuje wyłącznie obsługę błędów. Mam na myśli to, że niemal niemożliwe jest określenie, co ten kod tak naprawdę robi, ponieważ wszystko jest usiane obsługą błędów. Obsługa błędów jest ważna, *ale jeżeli utrudnia zrozumienie logiki kodu, jest niewłaściwa*.

W tym rozdziale przedstawię kilka technik i założeń, z których możemy skorzystać przy pisaniu kodu, który jest zarówno czytelny, jak i solidny — kodu, który obsługuje błędy elegancko i stylowo.

Użycie wyjątków zamiast kodów powrotu

W odległej przeszłości korzystaliśmy z wielu języków programowania nieposiadających wyjątków. W językach tych techniki obsługi i raportowania błędów były ograniczone. Można było ustawić znacznik błędu lub kod błędu, który następnie mógł być sprawdzany przez funkcję wywołującą. Podejście to jest przedstawione na listingu 7.1.

LISTING 7.1. DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Sprawdzenie stanu urządzenia.
    if (handle != DeviceHandle.INVALID) {
      // Zapisanie stanu urządzenia w polu rekordu.
      retrieveDeviceRecord(handle);
      // Jeżeli nie wstrzymane, wyłączenie.
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Urządzenie wstrzymane. Nie można wyłączyć");
      }
    } else {
      logger.log("Niewłaściwy uchwyt dla: " + DEV1.toString());
    }
  }
  ...
}
```

Rozwiązanie to jest jednak problematyczne, ponieważ powoduje bałagan w funkcji wywołującej. Musi ona sprawdzić błędy natychmiast po wykonaniu wywołania. Niestety, łatwo o tym zapomnieć. Z tego powodu w przypadku napotkania błędu lepiej zgłosić wyjątek. Kod wywołujący jest prostszy. Jego logika nie jest zanieczyszczona obsługą błędów.

Na listingu 7.2 przedstawiony jest wcześniejszy kod, w którym metody wykrywające błąd zgłaszają wyjątek.

LISTING 7.2. DeviceController.java (z wyjątkami)

```
public class DeviceController {
  ...
  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }

  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }
```

```
  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Niewłaściwy uchwyt dla: " + id.toString());
    ...
  }
  ...
}
```

Jak widać, ta wersja kodu jest znacznie czytelniejsza. To nie jest jednak wyłącznie kwestia estetyki. Kod jest lepszy, ponieważ dwa problemy, które były ze sobą splątane, algorytm wyłączenia urządzenia i obsługi błędów, zostały rozdzielone. Łatwiej teraz zrozumieć każdą z nich.

Rozpoczynanie od pisania instrukcji try-catch-finally

Jedną z najbardziej interesujących cech wyjątków jest to, że *definiują one zakresy* w naszym programie. Gdy wykonujemy kod w części `try` instrukcji `try-catch-finally`, zakładamy, że wykonanie może zostać w dowolnym momencie przerwane i wznowione w bloku `catch`.

Dzięki temu bloki `try` są podobne do transakcji. Nasz blok `catch` ma za zadanie pozostawić program w stanie spójnym, niezależnie od tego, co stało się w `try`. Z tego powodu dobrą praktyką jest korzystanie z instrukcji `try-catch-finally` w przypadku tworzenia kodu, w którym mogą zostać zgłoszone wyjątki. Pomaga to zdefiniować operacje, jakich powinien oczekiwać użytkownik tego kodu, niezależnie od tego, co złego wydarzy się w czasie wykonywania kodu z `try`.

Rozważmy następujący przykład. Chcemy napisać kod odwołujący się do pliku, z którego odczytuje serializowane obiekty.

Zaczynamy od testu jednostkowego, który pokazuje, że w przypadku braku pliku otrzymujemy wyjątek.

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

Test prowadzi nas do utworzenia następującego fragmentu:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // Zaślepka zwracanej wartości do momentu utworzenia implementacji.
  return new ArrayList<RecordedGrip>();
}
```

Test zawiedzie, ponieważ nie zgłasza wyjątku. Następnie zmieniamy naszą implementację, aby spróbować odwołać się do nieprawidłowego pliku. Operacja ta zgłasza wyjątek:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Test powiedzie się, ponieważ przechwyci wyjątek. Od tego momentu możemy zacząć modyfikowanie kodu. Możemy zawęzić typ przechwytywanego wyjątku, aby pasował do typu faktycznie zgłaszanego przez konstruktor `FileInputStream`: `FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Teraz, po zdefiniowaniu zakresu za pomocą struktury `try-catch`, możemy użyć TDD do zbudowania reszty potrzebnej logiki. Logika ta zostanie umieszczona pomiędzy utworzeniem `FileInputStream` a wywołaniem `close` i możemy w niej założyć, że wszystko przebiegnie prawidłowo.

Spróbujmy napisać testy wymuszające wyjątki, a następnie dodać do procedury obsługi operacje pozwalające spełnić te testy. Spowoduje to konieczność zbudowania zakresu transakcji za pomocą bloku `try`, co pomoże nam utrzymać transakcyjną naturę tego zakresu.

Użycie niekontrolowanych wyjątków

Debata jest zakończona. Przez lata programiści Javy debatowali nad zaletami i odpowiedzialnością kontrolowanych wyjątków. Gdy w pierwszej wersji języka Java zostały wprowadzone kontrolowane wyjątki, wydawało się to dobrym pomysłem. Sygnatura każdej metody powinna zawierać listę wszystkich wyjątków, jakie może przekazywać do wywołującego. Co więcej, wyjątki te były częścią typu metody. Kod po prostu nie skompilował się, jeżeli sygnatura nie odpowiadała temu, co robił kod.

Z czasem zaczęliśmy myśleć, że kontrolowane wyjątki były świetnym pomysłem — dawały one *pewne* korzyści. Obecnie jednak jest jasne, że nie są one potrzebne do tworzenia solidnego oprogramowania. C# nie posiada kontrolowanych wyjątków i, pomimo wielu prób, C++ też ich nie posiada. Nie ma ich również w językach Python czy Ruby. Jednak we wszystkich tych językach możliwe jest pisanie solidnego oprogramowania. Z powodu istnienia tak wielu przykładów musieliśmy zdecydować — naprawdę — czy kontrolowane wyjątki są warte ich ceny.

Jakiej ceny? Ceną kontrolowanych wyjątków jest naruszenie zasady otwarty-zamknięty[1]. Jeżeli zgłosimy kontrolowany wyjątek w metodzie naszego kodu, a instrukcja `catch` znajduje się trzy poziomy wyżej, *musimy zadeklarować ten wyjątek w sygnaturze każdej metody pomiędzy naszym kodem a `catch`*. Oznacza to, że zmiana wykonana na niskim poziomie oprogramowania wymusza zmiany na wielu wyższych poziomach. Zmienione moduły muszą zostać ponownie skompilowane i zainstalowane, nawet jeżeli zmiana bezpośrednio ich nie dotyczy.

[1] [Martin].

Weźmy pod uwagę hierarchię wywołań w dużym systemie. Funkcje znajdujące się na górnym poziomie wywołują funkcje znajdujące się poniżej, które wywołują kolejne funkcje poniżej, i tak dalej. Załóżmy teraz, że jedna z funkcji najniższego poziomu została zmodyfikowana w taki sposób, że musi zgłosić wyjątek. Jeżeli ten wyjątek jest kontrolowany, to do sygnatury funkcji należy dodać klauzulę `throws`. Jednak oznacza to, że każda funkcja, która wywołuje naszą zmodyfikowaną funkcję, musi być również zmodyfikowana, aby przechwytywała nowy wyjątek — w przeciwnym wypadku należy dołączyć do jej sygnatury klauzulę `throws`. Operacje te realizowane są w nieskończoność! Końcowym wynikiem są kaskadowe zmiany zaczynające się na najniższym poziomie oprogramowania i kończące na najwyższym! Naruszona jest również hermetyzacja, ponieważ wszystkie funkcje w ścieżce muszą poznać szczegóły tego niskopoziomowego wyjątku. Jako że przeznaczeniem wyjątków jest umożliwienie obsługi błędów na odległość, trzeba uznać za porażkę fakt, że kontrolowane wyjątki niszczą w ten sposób hermetyzację.

Kontrolowane wyjątki czasami mogą być użyteczne, na przykład gdy piszemy ważną bibliotekę. Konieczne jest wtedy ich przechwytywanie. Jednak w zwykłym programowaniu aplikacji koszt zależności jest większy niż ich zalety.

Dostarczanie kontekstu za pomocą wyjątków

Każdy zgłaszany przez nas wyjątek powinien mieć wystarczającą ilość informacji na temat kontekstu, aby można było określić źródło i lokalizację błędu. W języku Java możemy uzyskać ślad stosu z dowolnego wyjątku; jednak ślad stosu nie zawiera informacji o przeznaczeniu nieudanej operacji.

Trzeba tworzyć komunikaty błędów zawierające odpowiednie informacje i przekazywać je za pomocą wyjątków. Należy zamieścić w nich nieudaną operację i typ awarii. Jeżeli w naszej aplikacji stosujemy rejestrowanie, musimy przekazać wystarczające informacje, aby można było zarejestrować błąd w bloku `catch`.

Definiowanie klas wyjątków w zależności od potrzeb wywołującego

Istnieje wiele sposobów klasyfikowania błędów. Możemy klasyfikować je w zależności od źródła: czy pochodzą one z tego komponentu, czy innego? Lub w zależności od typu: czy jest to awaria urządzenia, sieci, czy błąd programowy? Jednak gdy definiujemy klasy wyjątków w aplikacji, największą uwagę powinniśmy zwrócić na *sposób ich przechwytywania*.

Spójrzmy na przykład słabej klasyfikacji wyjątków. Mamy tu instrukcję `try-catch-finally` do realizacji wywołania z zewnętrznej biblioteki. Obejmuje ona wszystkie wyjątki, jakie mogą zgłosić te wywołania:

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
```

```
    reportPortError(e);
    logger.log("Wyjątek odpowiedzi urządzenia", e);
  } catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Wyjątek odblokowania", e);
  } catch (GMXError e) {
    reportPortError(e);
    logger.log("Wyjątek odpowiedzi urządzenia");
  } finally {
    ...
  }
```

Ta instrukcja zawiera wiele duplikatów, ale nie powinniśmy być tym zaskoczeni. W większości przypadków obsługi wyjątków wykonywane operacje są względnie standardowe, niezależnie od faktycznej przyczyny. Musimy zarejestrować błąd i upewnić się, że możemy kontynuować.

W tym przypadku, ponieważ wiemy, że wykonywane operacje są w przybliżeniu identyczne, niezależnie od wyjątku, możemy znacznie uprościć tworzony kod przez otoczenie wywoływanego API i upewnienie się, że zwraca wspólny typ wyjątku:

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  ...
}
```

Nasza klasa `LocalPort` jest prostą klasą osłonową, przechwytującą i przekształcającą wyjątki zgłaszane przez klasę `ACMEPort`:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

Tego rodzaju klasy osłonowe, jakie zdefiniowaliśmy dla `ACMEPort`, mogą być bardzo przydatne. Faktycznie, osłanianie API firm zewnętrznych jest dobrą praktyką. Gdy osłaniamy API firmy zewnętrznej, minimalizujemy zależności od niego — w przyszłości można będzie zdecydować się na przejście na inną bibliotekę bez ponoszenia zbytnich kosztów. Osłanianie ułatwia również tworzenie imitacji klas zewnętrznych dla potrzeb testowania własnego kodu.

Ostatnią zaletą osłaniania jest to, że nie jesteśmy przywiązani do określonych decyzji projektowych producenta API. Można zdefiniować API, które będzie dla nas komfortowe. We wcześniejszym przykładzie zdefiniowaliśmy jeden wyjątek dla awarii urządzenia `port` i okazało się, że możemy pisać znacznie czytelniejszy kod.

Często jedna klasa wyjątków jest wystarczająca dla określonego obszaru kodu. Informacje wysyłane w wyjątku pozwalają nam rozróżniać błędy. Należy korzystać z różnych klas tylko w przypadkach, gdy chcemy przechwycić jeden wyjątek, a inny chcemy przepuścić.

Definiowanie normalnego przepływu

Jeżeli stosujemy się do porad z wcześniejszych punktów, uzyskamy wyraźny rozdział między logiką biznesową a obsługą błędów. Kod zaczyna wyglądać jak czysty, schludny algorytm. Jednak proces dochodzenia do takiej formy powoduje spychanie wykrywania błędów na margines naszego programu. Osłaniamy zewnętrzne API tak, aby zgłaszały wyjątki, i definiujemy procedury obsługi w kodzie, aby można było obsłużyć każdą przerwaną operację. W większości przypadków jest to świetne podejście, ale zdarzają się sytuacje, w których możemy nie chcieć przerywać pracy.

Rozważmy następujący przykład. Poniżej zamieszczony jest dziwny kod, w którym sumujemy wydatki w aplikacji księgowej:

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

W tej firmie wszystkie wydatki na posiłki wchodzą do podsumowania. Jeżeli nie będzie takiego wydatku, pracownicy otrzymują ryczałt dzienny. Wyjątek zaciemnia logikę kodu. Czy nie byłoby lepiej, gdybyśmy nie musieli obsługiwać specjalnego przypadku? Wówczas kod będzie znacznie prostszy. Mógłby wyglądać tak:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Czy będziemy w stanie tak uprościć kod? Okazuje się, że tak. Zmienimy klasę `ExpenseReportDAO` tak, aby zawsze zwracała obiekt `MealExpense`. Jeżeli nie będą występowały wydatki na posiłki, zwracany będzie obiekt `MealExpense` zawierający wartość ryczałtu dziennego:

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // Zwrócenie domyślnej wartości ryczałtu dziennego.
  }
}
```

Jest to przykład wzorca specjalnego przypadku (ang. *Special Case Pattern*) [Fowler]. Tworzymy w nim klasę lub konfigurujemy obiekt, aby obsługiwał szczególny przypadek za nas. Po takiej operacji kod klienta nie będzie musiał obsługiwać sytuacji wyjątkowej. Obsługa ta jest hermetyzowana w obiekcie przypadku specjalnego.

Nie zwracamy null

Uważam, że każde omówienie obsługi błędów powinno zawierać przedstawienie operacji, które powodują powstanie błędów. Pierwszą pozycją na liście jest zwracanie wartości `null`. Nie zliczę widzianych przeze mnie aplikacji, w których niemal każdy wiersz kodu zawierał test wartości `null`. Poniżej zamieszczony jest przykład takiego kodu:

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

Gdy zwracamy wartość `null`, w rzeczywistości tworzymy sobie dodatkową pracę i powodujemy problemy w funkcjach wywołujących. W takich przypadkach brak jednego testu wartości `null` powoduje, że aplikacja wymyka się spod kontroli.

Czy Czytelnik zauważył, że w drugim wierszu zagnieżdżonej instrukcji `if` nie ma kontroli wartości `null`? Co się stanie w trakcie działania programu, jeżeli `persistentStore` będzie `null`? Otrzymamy w takim przypadku wyjątek `NullPointerException` i albo ktoś przechwyci ten wyjątek na najwyższym poziomie, albo nie. W obu przypadkach jest to *złe*. Co można zrobić w odpowiedzi na wyjątek `NullPointerException` zgłoszony w głębi naszej aplikacji?

Łatwo powiedzieć, że problem w powyższym kodzie wynika z braku kontroli wartości `null`, ale w rzeczywistości problem polega na tym, że jest ich *zbyt dużo*. Jeżeli mamy zamiar zwrócić `null` z metody, warto rozważyć zgłoszenie wyjątku lub zwrócenie obiektu *specjalnego przypadku*. Jeżeli wywołujemy metodę zwracającą `null` pochodzącą z zewnętrznego API, warto rozważyć osłonięcie tej metody inną, która zgłasza wyjątek lub zwraca obiekt specjalnego przypadku.

W wielu sytuacjach obiekty specjalnych przypadków są prostym rozwiązaniem. Wyobraźmy sobie, że mamy następujący kod:

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Metoda `getEmployees` może zwrócić `null`, ale czy musi to robić? Jeżeli zmienimy `getEmployees` tak, aby zwracała pustą listę, możemy wyczyścić nasz kod:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

Na szczęście Java posiada metodę `Collections.emptyList()` zwracającą predefiniowaną niezmienną listę, która może być wykorzystana do takich celów.

```
public List<Employee> getEmployees() {
  if( .. brak pracowników .. )
    return Collections.emptyList();
}
```

Jeżeli będziemy tworzyć kod w taki sposób, zmniejszymy możliwość powstawania wyjątków `NullPointerException`, a nasz kod będzie czytelniejszy.

Nie przekazujemy null

Zwracanie wartości `null` z metod jest niedobrą praktyką, ale przekazywanie wartości `null` do metod jest jeszcze gorsze. O ile nie korzystamy z API, które oczekuje wartości `null`, i o ile mamy taką możliwość, powinniśmy unikać przekazywania `null` we własnym kodzie. Dlaczego? Rozważmy następujący przykład. Mamy tu prostą metodę, która oblicza odległość między dwoma punktami:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x – p1.x) * 1.5;
  }
  ...
}
```

Co się stanie, gdy ktoś przekaże `null` jako argument?

```
calculator.xProjection(null, new Point(12, 13));
```

Oczywiście otrzymamy `NullPointerException`.

Jak można to poprawić? Możemy utworzyć nowy typ wyjątku i zgłosić go:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Niewłaściwy argument dla MetricsCalculator.xProjection");
    }
    return (p2.x – p1.x) * 1.5;
  }
}
```

Czy tak jest lepiej? Być może jest to lepsze niż wyjątek wskaźnika `null`, ale należy pamiętać, że musimy zdefiniować procedurę obsługi zdarzenia `InvalidArgumentException`. Co powinna robić taka procedura? Czy istnieje właściwa akcja?

Istnieje inna opcja. Możemy skorzystać ze zbioru asercji.

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 nie może być null";
    assert p2 != null : "p2 nie może być null";
    return (p2.x – p1.x) * 1.5;
  }
}
```

Jest to dobra dokumentacja, ale nie rozwiązuje problemu. Jeżeli ktoś przekaże `null`, otrzyma błąd wykonania.

W większości języków programowania nie istnieje dobra metoda obsługi wartości `null` przypadkowo przekazanych przez wywołującą procedurę. Z tego powodu racjonalnym podejściem jest zakazanie przekazywania wartości `null`. W takim przypadku możemy tworzyć nasz kod, wiedząc, że `null` w liście argumentów jest symptomem problemu, co będzie skutkowało znacznie mniejszą liczbą pomyłek.

Zakończenie

Czysty kod jest czytelny, ale musi być również solidny. Nie są to cele pozostające w konflikcie. Możemy pisać solidny czysty kod, jeżeli będziemy traktować obsługę błędów jako osobne zadanie, coś, co jest widziane niezależnie od głównej logiki. Takie podejście do problemu ułatwi konserwację kodu w przyszłości.

Bibliografia

[Martin]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall 2002.

Granice

James Grenning

RZADKO KONTROLUJEMY WSZYSTKIE ELEMENTY KODU w naszym systemie. Czasami kupujemy pakiety firm zewnętrznych lub korzystamy z kodu open source. W innym przypadku polegamy na zespołach z naszej firmy, które produkują dla nas komponenty lub podsystemy. W jakiś sposób musimy w czysty sposób zintegrować ten obcy kod z naszym. W tym rozdziale zapoznamy się z praktykami i technikami pozwalającymi na zachowanie czystych granic naszego oprogramowania.

Zastosowanie kodu innych firm

Istnieje naturalne napięcie pomiędzy dostawcą interfejsu a jego użytkownikiem. Dostawcy pakietów i bibliotek starają się o szeroki zakres zastosowań, więc działają w wielu środowiskach i odwołują się do szerokiej gamy odbiorców. Z drugiej strony, użytkownicy żądają interfejsu, który jest przystosowany do ich konkretnych potrzeb. Takie napięcia mogą powodować problemy na granicach naszych systemów.

Rozważmy przykład klasy `java.util.Map`. Jak wynika z analizy rysunku 8.1, `Map` ma bardzo szeroki interfejs z wieloma możliwościami. Oczywiście, siła i elastyczność to przydatne cechy, ale mogą również wiązać się z dużą odpowiedzialnością. Nasze aplikacje mogą na przykład wykorzystywać obiekty `Map` oraz je przesyłać. Naszym założeniem może być, że żaden z odbiorców naszych obiektów `Map` nie będzie z nich nic usuwał. Jednak bezpośrednio na początku listy metod mamy `clear()`. Każdy użytkownik obiektu `Map` ma możliwość wyczyszczenia go. Być może nasza konwencja projektowa dyktuje, że w obiekcie `Map` można przechowywać obiekty określonego typu, ale obiekty `Map` nie mogą w niezawodny sposób ograniczać typów obiektów w nich umieszczanych. Każdy zdeterminowany użytkownik może dodawać do `Map` elementy dowolnego typu.

- clear() void - Map
- containsKey(Object key) boolean - Map
- containsValue(Object value) boolean - Map
- entrySet() Set - Map
- equals(Object o) boolean - Map
- get(Object key) Object - Map
- getClass() Class<? extends Object> - Object
- hashCode() int - Map
- isEmpty() boolean - Map
- keySet() Set - Map
- notify() void - Object
- notifyAll() void - Object
- put(Object key, Object value) Object - Map
- putAll(Map t) void - Map
- remove(Object key) Object - Map
- size() int - Map
- toString() String - Object
- values() Collection - Map
- wait() void - Object
- wait(long timeout) void - Object
- wait(long timeout, int nanos) void - Object

RYSUNEK 8.1. Metody klasy Map

Jeżeli nasza aplikacja potrzebuje obiektu `Map` dla obiektów `Sensor`, do tworzenia kontenera wykorzystamy następujący kod:

```
Map sensors = new HashMap();
```

Następnie, gdy w innej części kodu będziemy chcieli odwołać się do jego zawartości, użyjemy następującego kodu:

```
Sensor s = (Sensor)sensors.get(sensorId );
```

Nie będziemy używać tego wywołania jednokrotnie, ale wiele razy w całym kodzie. Klient tego kodu jest odpowiedzialny za pobranie obiektu `Object` z `Map` i rzutowanie go na właściwy typ. To działa, ale nie jest czystym kodem. Dodatkowo kod ten nie opowiada historii, tak jak powinien. Niezawodność tego kodu może być znacznie poprawiona przez zastosowanie typów ogólnych, jak poniżej:

```
Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId );
```

Nie rozwiązuje to jednak problemu dostarczania przez `Map<Sensor>` większych możliwości, niż potrzebujemy.

Przekazywanie obiektu `Map<Sensor>` w całym systemie powoduje, że w przypadku jakiejkolwiek zmiany interfejsu `Map` konieczne będzie wprowadzenie zmian w wielu miejscach. Można myśleć, że taka zmiana jest mało prawdopodobna, ale należy pamiętać, że już zdarzyła się po dodaniu obsługi typów ogólnych w Java 5. Faktycznie, spotkałem się z systemami, w których zrezygnowano z użycia typów ogólnych z powodu konieczności wprowadzenia olbrzymiej liczby zmian, które były wymagane, aby można było skorzystać z interfejsu `Map`.

Łatwiejszy sposób zastosowania `Map` jest przedstawiony poniżej. Żaden z użytkowników klasy `Sensors` nie będzie wiedział, czy typy ogólne zostały w niej użyte, czy nie. Wybór ten stał się (i zawsze był) szczegółem implementacji.

```
public class Sensors {
  private Map sensors = new HashMap();
  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }
// Odcinamy
}
```

Interfejs graniczny (`Map`) został ukryty. Jego ewolucja będzie miała niewielki wpływ na pozostałe części aplikacji. Zastosowanie typów ogólnych nie stanowi już dużego problemu, ponieważ rzutowanie i zarządzanie typami jest obsłużone wewnątrz klasy `Sensors`.

Interfejs ten jest również odpowiednio dostosowany i ograniczony, aby spełniał potrzeby aplikacji. Wynikowy kod jest przez to łatwiejszy do zrozumienia i trudniej popełnić pomyłkę. Klasa `Sensors` może wymuszać zasady projektowe i biznesowe.

Nie sugerujemy jednak, aby każde użycie `Map` było w ten sposób hermetyzowane. Zamiast tego radzimy nie przekazywać obiektów `Map` (ani innych interfejsów granicznych) po całym systemie. Jeżeli korzystamy z interfejsu granicznego, takiego jak `Map`, należy korzystać z niego w klasie lub w bliskiej rodzinie klas, w których jest wykorzystywany. Należy unikać zwracania go lub wykorzystywania jako argument w publicznym API.

Przeglądanie i zapoznawanie się z granicami

Kod firm zewnętrznych pozwala na dostarczenie większej liczby funkcji w krótszym czasie. Od czego zacząć, gdy będziemy chcieli wykorzystać obcy pakiet? Nie naszym zadaniem jest testowanie obcego kodu, ale w naszym interesie jest napisanie testów dla tego kodu, z którego korzystamy w projekcie.

Załóżmy, że nie jest jasne, jak używać pewnej biblioteki obcej firmy. Możemy spędzić dzień lub dwa (albo więcej), czytając dokumentację i podejmując decyzję, jak będziemy korzystać z tej biblioteki. Następnie możemy napisać nasz kod wykorzystujący kod firmy zewnętrznej i sprawdzić, czy realizuje to, czego się spodziewaliśmy. Nie będzie niespodzianką, gdy spędzimy długie godziny na debugowaniu i próbach zorientowania się, czy błędy znajdują się w naszym kodzie, czy obcym.

Uczenie się obcego kodu jest zawsze trudne. Integrowanie kodu obcego również jest trudne. Wykonywanie tych dwóch czynności jednocześnie jest podwójnie trudne. Być może warto spróbować innego podejścia? Zamiast eksperymentować z wykorzystaniem nowych funkcji w kodzie produkcyjnym, powinniśmy napisać kilka testów w celu skontrolowania naszego zrozumienia analizowanego kodu. Jim Newkirk nazywa to *testami uczącymi*[1].

W testach uczących wywołujemy API zewnętrzne tak, jak robilibyśmy to w naszej aplikacji. W rzeczywistości wykonujemy kontrolowane eksperymenty, sprawdzające naszą wiedzę na temat API. Testy skupiają się na elementach, które chcemy uzyskać z API.

Korzystanie z pakietu log4j

Załóżmy, że chcemy użyć pakietu `log4j` z Apache zamiast naszego własnego pakietu rejestrowania. Pobieramy go i otwieramy stronę wprowadzającą dokumentacji. Bez dokładnej lektury jesteśmy w stanie napisać nasz pierwszy test, w którym oczekujemy wypisania słowa *cześć* na konsoli.

```
@Test
public void testLogCreate() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.info("cześć");
}
```

Po jego uruchomieniu program zgłasza błąd informujący, że potrzebujemy czegoś o nazwie `Appender`. Po krótkiej lekturze znajdujemy informację, że dostępna jest klasa `ConsoleAppender`. Tworzymy więc `ConsoleAppender` i sprawdzamy, czy odkryliśmy sekrety kierowania komunikatów na konsolę.

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  ConsoleAppender appender = new ConsoleAppender();
  logger.addAppender(appender);
  logger.info("cześć");
}
```

[1] [BeckTDD], s. 136 – 137.

Tym razem okazuje się, że `Appender` nie posiada strumienia wyjściowego. Dziwne — wydaje się logiczne, że go mamy. Z niewielką pomocą Google próbujemy czegoś takiego:

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.removeAllAppenders();
  logger.addAppender(new ConsoleAppender(
    new PatternLayout("%p %t %m%n"),
    ConsoleAppender.SYSTEM_OUT));
  logger.info("cześć");
}
```

Tym razem program działa i na konsoli pojawia się komunikat *cześć*. Wydaje się dziwne, że musimy poinformować `ConsoleAdapter`, że ma pisać na konsolę.

Co ciekawe, gdy usuniemy argument `ConsoleAppender.SystemOut`, komunikat nadal będzie wyświetlany. Gdy usuniemy `PatternLayout`, program znów zacznie informować o braku strumienia wyjściowego. Jest to bardzo dziwne działanie.

Po dokładniejszym przyjrzeniu się dokumentacji zauważamy, że domyślny konstruktor `ConsoleAdapter` jest „nieskonfigurowany", co nie wydaje się zbyt oczywiste ani użyteczne. Wygląda to na błąd lub co najmniej na niekonsekwencję w pakiecie `log4j`.

Po dalszym szukaniu, czytaniu i testowaniu w końcu otrzymujemy kod z listingu 8.1. Wiele dowiedzieliśmy się na temat sposobu działania pakietu `log4j` i zawarliśmy tę wiedzę w zbiorze prostych testów jednostkowych.

LISTING 8.1. LogTest.java

```
public class LogTest {
    private Logger logger;

    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }

    @Test
    public void addAppenderWithStream() {
      logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
      logger.info("addAppenderWithStream");
    }
    @Test
    public void addAppenderWithoutStream() {
      logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n")));
      logger.info("addAppenderWithoutStream");
    }
}
```

Teraz wiemy już, jak należy zainicjować prosty system rejestrowania na konsoli, i możemy umieścić tę wiedzę we własnej klasie, aby pozostała część aplikacji była izolowana od interfejsu granicznego `log4j`.

Zalety testów uczących

Testy uczące nic nie kosztują. I tak musimy nauczyć się używanego API, a pisanie takich testów jest prostym sposobem uzyskania tej wiedzy. Testy uczące są precyzyjnymi eksperymentami, które pomagają nam zwiększyć wiedzę.

Testy uczące nie tylko są bezpłatne, ale również przynoszą wymierne korzyści. Gdy pojawi się nowa wersja pakietu zewnętrznego, uruchamiamy testy uczące i sprawdzamy, czy wystąpią różnice w działaniu.

Testy te pozwalają sprawdzić, czy używane przez nas obce pakiety działają tak, jak tego oczekujemy. Po zintegrowaniu obcego kodu z własnym nie mamy gwarancji, że ten zewnętrzny kod będzie nadal spełniał nasze oczekiwania. Jego autorzy mogli zmienić swój kod w taki sposób, aby spełniał inne oczekiwania. Poprawiają oni błędy i dodają nowe funkcje. Każda nowa wersja niesie ze sobą nowe zagrożenia. Jeżeli pakiet firmy zewnętrznej zmieni się w taki sposób, że będzie niezgodny z naszymi testami, natychmiast się o tym dowiemy.

Niezależnie od tego, czy potrzebujemy wiedzy zdobywanej za pomocą testów uczących, czy nie, powinniśmy pokryć kod graniczny testami, które sprawdzają interfejs w taki sam sposób, jak dzieje się to w kodzie produkcyjnym. Bez tych *testów granicznych* możemy np. korzystać ze starej wersji biblioteki dłużej, niż jest to potrzebne.

Korzystanie z nieistniejącego kodu

Istnieje również inny rodzaj granicy, która oddziela znane od nieznanego. Często spotykamy się z takimi fragmentami kodu, w których nasza wiedza jest wystawiona na próbę. Czasami to, co znajduje się po drugiej stronie granicy, jest nieznane (przynajmniej chwilowo). Czasami decydujemy się, aby nie zaglądać za granicę.

Kilka lat temu byłem członkiem zespołu tworzącego oprogramowanie dla systemu komunikacji radiowej. Istniał w nim podsystem „Transmitter", o którym niewiele wiedzieliśmy, a osoby odpowiedzialne za ten podsystem nie wpadły jeszcze na pomysł, aby zdefiniować interfejs. Nie chcieliśmy być przez to blokowani, więc zaczęliśmy pracę od odległych części kodu.

Wiedzieliśmy dokładnie, gdzie kończy się nasz, a zaczyna ich świat. W czasie naszej pracy czasami odbijaliśmy się od tej granicy. Pomimo że mgła i chmury ignorancji przesłaniały nasz widok poza granicę, w czasie pracy określiliśmy, jaki *chcielibyśmy* mieć interfejs graniczny. Chcieliśmy przekazać do transmitera mniej więcej taką informację:

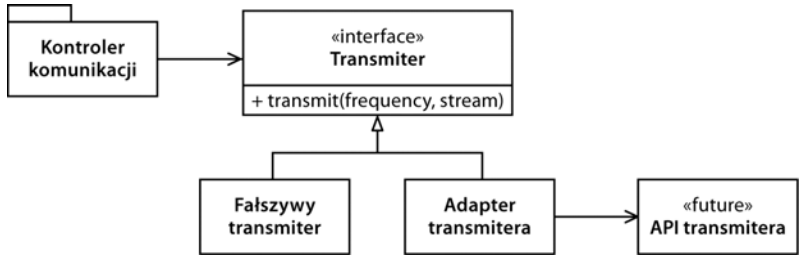
> *Ustaw transmiter na przekazanej częstotliwości i wyemituj analogową reprezentację danych przychodzących w tym strumieniu.*

Nie wiedzieliśmy, jak powinno być to zrobione, ponieważ API nie zostało jeszcze zaprojektowane. Zdecydowaliśmy więc popracować później nad szczegółami.

Aby nie blokować prac, zdefiniowaliśmy własny interfejs. Nadaliśmy mu jakąś chwytliwą nazwę, coś jak `Transmitter`. Dodaliśmy do niej metodę `transmit`, która oczekiwała przekazania częstotliwości i strumienia danych. Był to interfejs, którego *oczekiwaliśmy*.

Główną zaletą tworzenia oczekiwanego interfejsu jest to, że pozostaje on pod naszą kontrolą. Pozwala to na zachowanie większej czytelności kodu i ukierunkowanie go na wykonywaną akcję.

Na rysunku 8.2 można zobaczyć, że odizolowaliśmy klasy `CommunicationsController` od API transmitera (który był poza naszą kontrolą i nie był jeszcze zdefiniowany). Przez użycie interfejsu specyficznego dla interfejsu nasz kod `CommunicationsController` pozostał czysty i komunikatywny. Po zdefiniowaniu API transmitera napisaliśmy klasę `TransmitterAdapter` łatającą dziurę. Adapter[2] hermetyzował interakcję z API i zapewniał jedno miejsce zmian w przypadku ewolucji API.



RYSUNEK 8.2. Przewidywanie klas transmitera

Projekt ten pozwolił na utworzenie bardzo wygodnego szwu[3] w kodzie, co ułatwia testowanie. Przy użyciu odpowiedniej klasy `FakeTransmitter` mogliśmy przetestować klasy `CommunicationsController`. Mogliśmy również utworzyć testy graniczne po uzyskaniu `TransmitterAPI`, aby upewnić się, że prawidłowo korzystamy z tego API.

Czyste granice

Na granicach dzieją się interesujące zdarzenia. Wśród nich są m.in. zmiany. Dobry projekt oprogramowania przyjmuje zmiany bez ogromnych nakładów i ponownej pracy. Gdy korzystamy z kodu pozostającego poza naszą kontrolą, musimy użyć specjalnych metod w celu ochrony naszych inwestycji, które zapewniają, że przyszłe modyfikacje nie będą zbyt kosztowne.

Kod na tych granicach wymaga jasnej separacji i testów definiujących oczekiwania. Powinniśmy unikać sytuacji, w której zbyt wiele naszego kodu korzysta ze szczegółów zdefiniowanych w obcym kodzie. Lepiej polegać na czymś, co *my* możemy kontrolować, niż na czymś, czego nie możemy kontrolować, ponieważ kończy się to... kontrolowaniem nas.

---

[2] Patrz wzorzec Adapter w [GOF].

[3] Więcej informacji na temat szwów można znaleźć w [WELC].

Zarządzamy granicami z obcym kodem przez ograniczenie do minimum miejsc w kodzie odwołujących się do niego. Możemy hermetyzować ten kod tak, jak zrobiliśmy to w `Map`, lub użyć adaptera w celu konwersji naszego perfekcyjnego interfejsu na dostarczony interfejs. W obu przypadkach kod lepiej do nas przemawia, promuje wewnętrzne spójne nazewnictwo na granicach i posiada mniej punktów konserwacji w przypadku zmiany zewnętrznego kodu.

Bibliografia

[BeckTDD]: Kent Beck, *Test Driven Development*, Addison-Wesley 2003.

[GOF]: Gamma, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

[WELC]: *Working Effectively with Legacy Code*, Addison-Wesley 2004.

ROZDZIAŁ 9.

Testy jednostkowe

NASZA PROFESJA PRZESZŁA ZNACZNE ZMIANY w czasie ostatnich dziesięciu lat. W roku 1997 nikt nie słyszał o programowaniu sterowanym testami (ang. *Test Driven Development*). Dla znacznej większości z nas testy jednostkowe były krótkimi fragmentami kodu, które pisaliśmy w celu upewnienia się, że nasze programy działają. Z mozołem pisaliśmy klasy i metody, a następnie tworzyliśmy fragmenty kodu do ich przetestowania. Zwykle wymagało to użycia jakiegoś programu sterującego, który pozwalał ręcznie sterować napisanym kodem.

W latach dziewięćdziesiątych pisałem program w języku C++ dla wbudowanego systemu czasu rzeczywistego. Program był prostym stoperem o następującej sygnaturze:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

Pomysł był dosyć prosty, metoda `execute` obiektu `Command` wywoływała nowy wątek po zdefiniowanej liczbie milisekund. Powstał problem, jak go przetestować.

Skleciłem więc prosty program sterujący, który nasłuchiwał zdarzeń klawiatury. Po każdym wpisaniu znaku planowane było wykonanie polecenia wpisującego ten sam znak pięć sekund później. Następnie wystukiwałem rytmiczną melodię na klawiaturze i melodia ta była odtwarzana na ekranie po pięciu sekundach.

„I … want-a-girl … just … like-the-girl-who-marr … ied … dear … old … dad".

Faktycznie nuciłem tę melodię, naciskając klawisz „.", a potem ponownie ją nuciłem, gdy kropki pojawiały się na ekranie.

To był mój test! Gdy sprawdziłem, że kod działa, i pokazałem to moim kolegom, skasowałem kod testu.

Jak powiedziałem, nasza profesja przeszła znaczne zmiany. Obecnie napisałbym test, który upewniłby mnie, że każdy fragment i zakątek kodu działa tak, jak tego oczekuję. Wyizolowałbym swój kod z systemu operacyjnego, zamiast wywoływać standardowe funkcje stopera. Użyłbym imitacji tych funkcji stopera, aby mieć pełną kontrolę nad czasem. Zaplanowałbym polecenia, które ustawiałyby znaczniki, a następnie po upływie określonego czasu kontrolowałbym te znaczniki, upewniając się, że zmieniły swój stan na właściwy dla czasu, jaki bym ustawił.

Gdy testy byłyby wykonywane prawidłowo, upewniłbym się, że może je bez trudu uruchomić każdy, kto będzie chciał korzystać z mojego kodu. Upewniłbym się, że testy i sprawdzany przez nie kod znajdują się w tym samym pakiecie kodu źródłowego.

Tak, przeszliśmy długą drogę, ale musimy pójść jeszcze dalej. Metodologie Agile oraz TDD zachęciły wielu programistów do pisania automatycznych testów jednostkowych i każdego dnia ich liczba rośnie. Jednak w owczym pędzie do korzystania z testów wielu programistów nie zdaje sobie sprawy z kilku bardziej subtelnych i istotnych cech dobrych testów.

Trzy prawa TDD

Obecnie wszyscy wiedzą, że TDD zakłada pisanie testów jednostkowych na początku, przed napisaniem kodu produkcyjnego. Jednak zasada ta jest tylko wierzchołkiem góry lodowej. Możemy zdefiniować następujące trzy prawa[1]:

Pierwsze prawo. Nie można zacząć pisać kodu produkcyjnego do momentu napisania niespełnianego testu jednostkowego.

Drugie prawo. Nie można napisać więcej testów jednostkowych, które są wystarczające do niespełnienia testu, a brak kompilacji jest jednocześnie nieudanym testem.

---

[1] Robert C. Martin, *Professionalism and Test-Driven Development*, Object Mentor, IEEE Software, maj – czerwiec 2007 (Vol. 24, No. 3), s. 32 – 36, *http://doi.ieeecomputersociety.org/10.1109/MS.2007.85*.

Trzecie prawo. Nie można pisać większej ilości kodu produkcyjnego, niż wystarczy do spełnienia obecnie niespełnianego testu.

Te trzy prawa zamykają nas w cyklu, który trwa prawdopodobnie trzydzieści sekund. Testy i kod produkcyjny są pisane *razem*, a testy są wykonywane kilka sekund wcześniej niż kod produkcyjny.

Jeżeli będziemy w ten sposób pracować, napiszemy kilkanaście testów każdego dnia, setki każdego miesiąca i tysiące testów co rok. Testy te będą pokrywać niemal cały kod produkcyjny. Ogromna liczba tych testów, które będą podobnej objętości co kod produkcyjny, może prowadzić do poważnych problemów z zarządzaniem projektem.

Zachowanie czystości testów

Kilka lat temu zostałem poproszony o kierowanie zespołem, który wcześniej zdecydował, że jego kod testujący *nie powinien* być utrzymywany zgodnie z tymi samymi standardami jakości co kod produkcyjny. Programiści dali sobie przyzwolenie na łamanie reguł w testach jednostkowych. Myślą przewodnią było „szybki i brudny". Zmienne nie musiały mieć odpowiednich nazw, funkcje testujące nie musiały być krótkie i opisowe. Ich kod testujący nie musiał być odpowiednio zaprojektowany i właściwie podzielony. Dopóki kod testu działał i pokrywał kod produkcyjny, był wystarczająco dobry.

Część Czytelników może sympatyzować z tymi decyzjami. Być może w dalekiej przeszłości pisali oni testy podobne do mojego testu klasy `Timer`. Jednak pomiędzy tego rodzaju odrzucanymi testami a automatycznymi testami jednostkowymi istnieje ogromna przepaść. Tak więc podobnie w kierowanym przeze mnie zespole moglibyśmy zdecydować, że posiadanie „brudnych" testów jest lepsze niż ich brak.

Jednak zespół ten nie zorientował się, że korzystanie z takich testów odpowiada... brakowi testów, o ile nie jest czymś gorszym. Problem wynika z tego, że testy muszą się zmieniać wraz z ewoluowaniem kodu produkcyjnego. Im gorsze są testy, tym trudniej je zmieniać. Im bardziej zagmatwany jest kod testujący, tym większe prawdopodobieństwo, że spędzimy więcej czasu na dodawaniu nowych testów do pakietu niż na pisaniu nowego kodu produkcyjnego. Wraz z modyfikowaniem kodu produkcyjnego stare testy przestają się wykonywać, a bałagan w kodzie testującym utrudnia doprowadzenie ich do stanu zgodnego z kodem produkcyjnym. Dlatego testy zaczynają być postrzegane jako stale zwiększająca się odpowiedzialność.

W kolejnych wersjach koszt utrzymania zestawu testów w moim zespole stale wzrastał. W końcu stał się największą uciążliwością dla programistów. Gdy kierownicy pytali, dlaczego ich oszacowania są tak duże, programiści winili za to testy. W końcu zostali zmuszeni do całkowitego usunięcia zestawu testów.

Jednak bez zestawu testów utraciliśmy możliwość upewnienia się, że zmiany w bazie kodu działają w oczekiwany sposób. Bez zestawu testów nie mogliśmy upewnić się, że zmiany w jednej części systemu nie spowodują błędu w innych jego częściach. Dlatego liczba defektów zaczęła wzrastać.

Gdy liczba nieoczekiwanych defektów zaczęła wzrastać, programiści zaczęli obawiać się wprowadzania zmian. Przestali czyścić swój kod produkcyjny, ponieważ obawiali się, że zmiany wprowadzą więcej szkody niż pożytku. Ich kod produkcyjny zaczął się psuć. W końcu zostali bez testów, z zagmatwanym i zawierającym błędy kodem produkcyjnym, sfrustrowanymi klientami oraz poczuciem, że ich praca nad testami poszła na marne.

W pewnym stopniu mieli rację. Ich praca przy testach *zawiodła*. Jednak ich decyzja o pozwoleniu na tworzenie zagmatwanych testów była zalążkiem porażki. Gdyby ich testy były utrzymywane w odpowiednim stanie, praca przy testach nie poszłaby na marne. Mogę to powiedzieć z dużą pewnością, ponieważ brałem udział w projektach i kierowałem wieloma zespołami, które z dużym sukcesem korzystały z *czystych* testów jednostkowych.

Morał z tej historii jest prosty: *kod testów jest tak samo ważny jak kod produkcyjny*. Nie jest to obywatel drugiej kategorii. Wymaga przemyślenia, zaprojektowania i uwagi. Musi być utrzymywany zgodnie z takimi samymi zasadami co kod produkcyjny.

Testy zwiększają możliwości

Jeżeli nie będziemy zachowywali czystości testów, utracimy je. Bez nich utracimy jedyną rzecz, która zapewnia elastyczność kodu produkcyjnego. Tak, dobrze przeczytaliście. To właśnie *testy jednostkowe* zapewniają elastyczność, łatwość utrzymania i ponownego wykorzystania kodu. Powód jest prosty. Jeżeli mamy testy, nie musimy bać się wprowadzania zmian do kodu! Bez testów każda zmiana jest potencjalnym błędem. Niezależnie od tego, jak elastyczna jest nasza architektura, niezależnie od tego, jak dobrze partycjonowany jest nasz projekt, bez testów będziemy obawiali się wprowadzać zmiany z powodu strachu przed wprowadzeniem niewykrytych błędów.

Jednak gdy testy są *używane*, obawa ta znika niemal zupełnie. Im lepsze jest pokrycie kodu testami, tym obawa jest mniejsza. Można niemal bezkarnie wprowadzać zmiany do kodu, którego architektura nie jest doskonała, a projekt jest zagmatwany i niejasny. Można nawet bez obawy *poprawiać* tę architekturę i projekt!

Tak więc posiadanie zestawu testów automatycznych obejmujących kod produkcyjny jest kluczowe w utrzymaniu projektu i architektury w możliwie najlepszym stanie. Testy znacznie zwiększają możliwości, ponieważ pozwalają na *zmiany*.

Jeżeli nasze testy są zagmatwane, to możliwości zmiany kodu są zmniejszone i zaczynamy tracić możliwość poprawiania struktury kodu. Im gorsze są nasze testy, tym bardziej zagmatwany staje się kod. W końcu tracimy testy, a nasz kod zaczyna się psuć.

Czyste testy

Co sprawia, że testy są czyste? Trzy elementy: czytelność, czytelność i jeszcze raz czytelność. Czytelność jest prawdopodobnie ważniejsza w przypadku testów jednostkowych niż w kodzie produkcyjnym. Co sprawia, że testy są czytelne? Te same elementy, które powodują, że każdy

kod jest czytelny — klarowność, prostota i spójność kodu. W teście chcemy wyrazić możliwie dużo przy użyciu możliwie niewielu wyrażeń.

Przeanalizujmy kod z listingu 9.1, pochodzący z FitNesse. Testy te są trudne do zrozumienia i powinny być poprawione. Po pierwsze, mamy tu ogromną ilość powtarzanego kodu [G5] w powtarzanych wywołaniach `addPage` oraz `assertSubString`. Co ważniejsze, kod ten jest przeładowany szczegółami, które niekorzystnie wpływają na jasność wyrażeń w testach.

LISTING 9.1. SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
      new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
  WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties();
  WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
      new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception
{
  crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");
```

```
    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
      (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
  }
```

Dla przykładu spójrzmy na wywołania `PathParser`. Przekształcają one ciąg znaków w obiekt `PagePath` wykorzystywany przez mechanizm przeszukiwania. Transformacja ta jest niepotrzebna do testowania i jedynie zaciemnia nasze intencje. Szczegóły konieczne do utworzenia obiektu `responder` oraz zbierania i rzutowania `response` są również tylko szumem. Następnie znajduje się tam mechanizm budowania URL żądania na podstawie obiektu `resource` oraz argumentu (pomagałem pisać ten kod, więc mogę swobodnie go krytykować).

W końcu kod ten nie był zaprojektowany do czytania. Biedny Czytelnik zostanie zarzucony wieloma szczegółami, jakie musi zrozumieć, aby testy miały sens.

Popatrzmy teraz na ulepszone testy z listingu 9.2. Testy te wykonują dokładnie te same zadania, ale zostały przebudowane do czystszej i bardziej zrozumiałej postaci.

LISTING 9.2. SerializedPageResponderTest.java (przebudowany)

```
  public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
  }

  public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
    assertResponseDoesNotContain("SymPage");
  }

  public void testGetDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");

    submitRequest("TestPageOne", "type:data");

    assertResponseIsXML();
    assertResponseContains("test page", "<Test");
  }
```

Struktura tych testów powoduje, że wzorzec *BUILD-OPERATE-CHECK*[2] jest oczywisty. Każdy z tych testów jest podzielony na trzy części. Pierwsza część buduje dane testowe, druga operuje na danych testowych, a trzecia kontroluje, by operacja dała oczekiwane wyniki.

Należy zwrócić uwagę, że większość irytujących szczegółów została wyeliminowana. Testy idą w dobrym kierunku i korzystają tylko z tych typów danych i funkcji, które są faktycznie potrzebne. Każdy, kto czyta te testy, powinien móc zorientować się, co się dzieje, bez konieczności analizowania mnóstwa szczegółów.

Języki testowania specyficzne dla domeny

Testy z listingu 9.2 demonstrują technikę budowania języka specyficznego dla domeny na potrzeby naszych testów. Zamiast korzystać z API stosowanego przez programistów do manipulowania systemem, budujemy zbiór funkcji i narzędzi wykorzystujących to API, które pozwalają na wygodniejsze pisanie testów, które z kolei są łatwiejsze do czytania. Te funkcje i narzędzia stają się specjalizowanym API wykorzystywanym przez testy. Są one *językiem* testowania wykorzystywanym przez programistów przy pisaniu testów oraz wspierającym osoby czytające później testy.

Takie API nie jest zaprojektowane od początku; raczej ewoluowało ze stale przebudowywanego kodu testowego, który stał się zbyt zatłoczony przez szczegóły. Podobnie jak przebudowaliśmy kod z listingu 9.1 na ten z listingu 9.2, zdyscyplinowani programiści przebudowują swój kod testowy na bardziej zwięzłe i czytelne formy.

Podwójny standard

Wspomniany przeze mnie na początku tego rozdziału zespół miał rację w jednej sprawie. Kod w API testowym *faktycznie* ma inny zbiór standardów inżynierskich niż kod produkcyjny. Musi nadal być prosty, zwięzły i czytelny, ale nie musi być tak wydajny jak kod produkcyjny. W końcu działa on w środowisku testowym, a nie produkcyjnym, a te dwa środowiska mają całkiem inne wymagania.

Przeanalizujmy kod z listingu 9.3. Napisałem te testy jako część systemu kontrolowania środowiska, którego prototyp opracowałem. Bez wchodzenia w szczegóły można powiedzieć, że testy te kontrolują, czy alarm niskiej temperatury, ogrzewacz i wentylator są włączone, gdy temperatura jest „wyraźnie zbyt niska".

LISTING 9.3. EnvironmentControllerTest.java

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

[2] *http://fitnesse.org/FitNesse.AcceptanceTestPatterns*

Mamy tu oczywiście wiele szczegółów. Na przykład do czego służy funkcja `tic`? W rzeczywistości nie przejmowałbym się tym w czasie czytania tych testów. Obawiałbym się raczej tego, czy końcowy stan systemu jest spójny, gdy temperatura jest „wyraźnie zbyt niska".

Należy zauważyć, że gdy czytamy testy, nasze oko musi skakać pomiędzy nazwą sprawdzanego stanu a *czujnikiem* sprawdzanego stanu. Widzimy `heaterState` i nasze oko prześlizguje się w lewo do `assertTrue`. Widzimy `coolerState` i nasze oko musi skierować się w lewo do `assertFalse`. Jest to nużące i zawodne. Powoduje, że testy są trudne do czytania.

Znacznie poprawiłem czytelność tych testów przez przekształcenie ich na postać z listingu 9.4.

LISTING 9.4. EnvironmentControllerTest.java (przebudowany)

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

Oczywiście, ukryłem szczegóły funkcji `tic` przez utworzenie funkcji `wayTooCold`. Zauważamy jednak dziwny ciąg znaków w `assertEquals`. Wielka litera oznacza „włączony", a mała „wyłączony"; litery są zawsze w następującej kolejności: `{heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}`.

Pomimo że jest to bliskie naruszenia zasady odwzorowania mentalnego[3], w tym przypadku wydaje się działaniem właściwym. Należy zauważyć, że gdy znamy znaczenie, nasze oko prześlizguje się po tym ciągu znaków i możemy szybko zinterpretować wyniki. Czytanie testów staje się niemal przyjemnością. Spójrzmy na listing 9.5 i przekonajmy się, jak łatwo można zrozumieć znajdujące się tam testy.

LISTING 9.5. EnvironmentControllerTest.java (większy fragment)

```
@Test
  public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
  }

  @Test
  public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
  }

  @Test
  public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
  }

  @Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

[3] „Unikanie odwzorowania mentalnego" w rozdziale 2.

Funkcja `getState` jest zamieszczona na listingu 9.6. Warto zwrócić uwagę, że kod ten nie jest zbyt efektywny. Aby poprawić jego efektywność, prawdopodobnie musiałbym użyć klasy `StringBuffer`.

LISTING 9.6. MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}
```

Zastosowanie `StringBuffer` jest mało eleganckie. Nawet w kodzie produkcyjnym unikam ich, jeżeli koszt jest niewielki, a można się zgodzić, że koszt w kodzie z listingu 9.6 jest niewielki. Jednak aplikacja ta wchodzi w skład wbudowanego systemu czasu rzeczywistego, a najprawdopodobniej zasoby komputera i pamięć są bardzo ograniczone. Środowisko *testowe* nie jest jednak tak ograniczone.

Taka jest natura podwójnych standardów. Istnieją operacje, których nigdy nie użylibyśmy w środowisku produkcyjnym, ale są całkowicie dozwolone w środowisku testowym. Zwykle jest to związane z efektywnością obliczeniową i pamięciową. Jednak *nigdy* nie są to problemy z czystością kodu.

Jedna asercja na test

Istnieje szkoła programowania[4] twierdząca, że każda funkcja testowa w JUnit powinna mieć jedną instrukcję asercji. Zasada ta może wydawać się drakońska, ale jej zalety są widoczne na listingu 9.5. Poszczególne testy zawierają pojedynczą konkluzję, którą można szybko i łatwo zrozumieć.

Co jednak z listingiem 9.2? Wydaje się nierozsądne, że możemy dosyć łatwo łączyć asercje, których wynikiem jest XML i które zawierają określone podciągi. Możemy jednak podzielić test na dwa osobne testy, z których każdy będzie miał własną asercję, tak jak na listingu 9.7.

LISTING 9.7. SerializedPageResponderTest.java (pojedyncze asercje)

```
public void testGetPageHierarchyAsXml() throws Exception {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
  whenRequestIsIssued("root", "type:pages");
  thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
  whenRequestIsIssued("root", "type:pages");
  thenResponseShouldContain(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
}
```

[4] Patrz wpis na blogu Dave'a Astela: *http://www.artima.com/weblogs/viewpost.jsp?thread=35578*.

Zwróćmy uwagę, że zmieniłem nazwy funkcji, aby korzystały z często stosowanej konwencji *given-when-then*[5]. Powoduje to, że testy są jeszcze łatwiejsze w czytaniu. Niestety, tego typu rozdzielenie testów powoduje powstanie dużej ilości powtórzonego kodu.

Możemy wyeliminować te powtórzenia przez zastosowanie wzorca szablonu metody[6] (ang. *Template Method*) i umieszczenie części *given-then* w klasie bazowej, a części *then* w poszczególnych klasach pochodnych. Można również utworzyć całkowicie osobne klasy testów i umieścić części *given* oraz *then* w funkcji `@Before`, a części *when* w każdej funkcji `@Test`. Jednak wydaje się nieracjonalne użycie tak wielu mechanizmów do rozwiązania niewielkiego problemu. Osobiście najbardziej lubię wielokrotne asercje z listingu 9.2.

Uważam, że zasada jednej asercji jest dobrą wskazówką[7]. Zwykle staram się tworzyć specyficzny dla domeny język testowania, który je wspiera, tak jak w przypadku kodu z listingu 9.5. Jednak nie obawiam się umieszczać więcej niż jednej asercji w teście. Uważam, że możemy jedynie stwierdzić, iż liczba asercji w teście powinna być zminimalizowana.

Jedna koncepcja na test

Lepszą zasadą dotyczącą testów jest obejmowanie jednej koncepcji w każdej funkcji testowej. Nie potrzebujemy długich funkcji testowych, które kolejno testują jedną rzecz za drugą. Przykład takiego testu jest zamieszczony na listingu 9.8. Test ten powinien być podzielony na trzy niezależne testy, ponieważ sprawdzane są trzy niezależne kwestie. Złączenie ich w tej samej funkcji wymusza na Czytelniku konieczność sprawdzania, w której sekcji się znajduje i co jest w niej testowane.

LISTING 9.8.

```
/**
 * Różne testy metody addMonths().
 */
public void testAddMonths() {
  SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

  SerialDate d2 = SerialDate.addMonths(1, d1);
  assertEquals(30, d2.getDayOfMonth());
  assertEquals(6, d2.getMonth());
  assertEquals(2004, d2.getYYYY());

  SerialDate d3 = SerialDate.addMonths(2, d1);
  assertEquals(31, d3.getDayOfMonth());
  assertEquals(7, d3.getMonth());
  assertEquals(2004, d3.getYYYY());

  SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
  assertEquals(30, d4.getDayOfMonth());
  assertEquals(7, d4.getMonth());
  assertEquals(2004, d4.getYYYY());
}
```

---

[5] [RSpec].

[6] [GOF].

[7] „Trzymaj się kodu!".

Te trzy funkcje testowe powinny prawdopodobnie wyglądać następująco:

- *Mamy* ostatni dzień miesiąca mającego 31 dni (tak jak maj):
  1. *Gdy* dodamy jeden miesiąc, tak że ostatnim dniem tego miesiąca jest 30 (tak jak czerwiec), *to* data powinna być 30 tego miesiąca, a nie 31.
  2. *Gdy* dodamy do tej daty dwa miesiące, tak aby wynikowy miesiąc miał 31 dni, *to* data powinna być 31.
- *Mamy* ostatni dzień miesiąca mającego 30 dni (tak jak czerwiec):
  1. *Gdy* dodamy jeden miesiąc, tak aby ostatni dzień tego miesiąca miał 31 dni, *to* wynikowa data powinna być 30, a nie 31.

Na podstawie takich założeń można zauważyć, że istnieje ogólna zasada ukrywania testów. Gdy zwiększymy miesiąc, data nie może być większa niż ostatni dzień miesiąca. Powoduje to, że zwiększenie miesiąca w przypadku 28 lutego powinno dać w wyniku 20 marca. *Tego* testu brakuje, a byłby on przydatny.

Tak więc to nie wielokrotne asercje z listingu 9.8 powodują problem. Problemem jest raczej testowanie więcej niż jednej koncepcji. Dlatego najlepszą zasadą jest minimalizacja liczby asercji dla koncepcji i testowanie tylko jednej koncepcji na funkcję testową.

F.I.R.S.T.[8]

Czyste testy powinny spełniać pięć innych zasad, które tworzą powyższy akronim:

Szybkie (Fast). Testy powinny być szybkie. Powinny działać szybko. Gdy testy działają powoli, nie chcemy ich uruchamiać zbyt często. Jeżeli nie uruchamiamy ich zbyt często, nie znajdziemy problemów wystarczająco szybko, aby można było łatwo je poprawić. Nie czujemy się wystarczająco pewnie, aby czyścić nasz kod. W końcu kod zaczyna się psuć.

Niezależne (Independent). Testy nie powinny zależeć od siebie. Jeden test nie powinien konfigurować warunków do następnego testu. Powinniśmy być w stanie uruchamiać każdy test niezależnie i uruchamiać testy w dowolnie wybranym porządku. Jeśli testy zależą od siebie, to gdy nie uda się pierwszy test, powstaje kaskada awarii, co utrudnia diagnozę i ukrywa awarie na niższym poziomie.

Powtarzalne (Repeatable). Testy powinny być powtarzalne w każdym środowisku. Powinniśmy być w stanie uruchomić testy w środowisku produkcyjnym, środowisku zapewnienia jakości oraz na naszym laptopie w trakcie podróży pociągiem do domu. Jeżeli nasze testy nie są powtarzalne w każdym środowisku, to zawsze będziemy mieli wymówkę, gdy się nie powiodą. Okazuje się również, że możemy mieć kłopoty z uruchomieniem testów, gdy nie jest dostępne dane środowisko.

---

[8] Materiały szkoleniowe mentora obiektowego.

Samokontrolujące się (Self-Validating). Testy powinny mieć jeden parametr wyjściowy typu logicznego. Mogą one się powieść lub nie. Nie powinniśmy musieć czytać plików dzienników w celu sprawdzenia, czy testy się powiodły. Nie powinniśmy ręcznie porównywać dwóch plików testowych w celu sprawdzenia, czy test się powiódł. Jeżeli testy nie kontrolują się samodzielnie, to awaria może stać się subiektywna i uruchomienie testów będzie wymagać długiego procesu ręcznej analizy.

O czasie (Timely). Testy powinny być pisane w odpowiednim momencie. Testy jednostkowe powinny być pisane *bezpośrednio przed* tworzeniem testowanego kodu produkcyjnego. Jeżeli piszemy testy po kodzie produkcyjnym, może się okazać, że jest on trudny do przetestowania. Można zdecydować, że pewna część kodu produkcyjnego jest zbyt trudna do przetestowania. Nie można zaprojektować kodu produkcyjnego tak, aby nie dał się przetestować.

Zakończenie

Niniejszy rozdział stanowi zaledwie zarys problematyki testowania kodu. Faktycznie, można napisać całą książkę na temat *czystych testów*. Testy są tak ważne dla jakości projektu, jak sam kod produkcyjny. Prawdopodobnie są nawet ważniejsze, ponieważ zapewniają i rozszerzają elastyczność, łatwość utrzymania i ponownego użycia kodu produkcyjnego. Tak więc testy powinny być stale utrzymywane w czystości. Należy zapewnić, że będą czytelne i zwięzłe. Warto opracować API testujące, które posłuży jako język specyficzny dla domeny, pomagający w pisaniu testów.

Jeżeli pozwolimy na zepsucie testów, nasz kod również będzie się psuł. Testy należy utrzymywać w czystości.

Bibliografia

[RSpec]: Aslak Hellesøy, David Chelimsky, *RSpec: Behavior Driven Development for Ruby Programmers*, Pragmatic Bookshelf 2008.

[GOF]: Gamma, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

Klasy

we współpracy z Jeffem Langrem

DO TEJ PORY W KSIĄŻCE skupialiśmy się na odpowiednim zapisywaniu wierszy i bloków kodu. Zajmowaliśmy się prawidłową kompozycją funkcji oraz ich wzajemnymi relacjami. Jednak pomimo całego zaangażowania w zapewnienie ekspresywności instrukcji kodu, jak również zbudowanych z niego funkcji, nadal nie mamy czystego kodu; musimy poświęcić uwagę wyższym poziomom organizacji kodu. Zajmijmy się czystością klas.

Organizacja klas

Zgodnie ze standardową konwencją języka Java klasy powinny zaczynać się od listy zmiennych. Publiczne stałe statyczne, jeżeli występują, powinny być zapisane jako pierwsze. W dalszej kolejności zapisywane są prywatne zmienne statyczne, a po nich prywatne zmienne instancyjne. Rzadko istnieje dobry powód korzystania ze zmiennych publicznych.

Po liście zmiennych powinny znajdować się funkcje publiczne. Prywatne funkcje użytkowe wywoływane przez funkcje publiczne zwykle umieszczamy po funkcjach publicznych. Jest to zgodne z zasadą zstępującą i pomaga czytać program jak artykuł w gazecie.

Hermetyzacja

Zazwyczaj nasze zmienne i funkcje użytkowe pozostają prywatne, ale nie jesteśmy fanatykami tego rozwiązania. Czasami zmieniamy zmienną lub funkcję na chronioną, aby była dostępna dla testu. Dla nas to testy ustalają zasady. Jeżeli test w tym samym pakiecie potrzebuje wywołać funkcję lub odwołać się do zmiennej, zmieniamy jej zasięg na chroniony lub dostępny w ramach pakietu. Jednak na początku zawsze szukamy sposobu na zachowanie prywatności. Rozluźnianie zasady prywatności jest ostatnim możliwym rozwiązaniem.

Klasy powinny być małe!

Pierwsza zasada dotycząca klas mówi, że powinny być małe. Druga zasada — że *powinny być mniejsze, niż są*. Nie, nie mamy zamiaru powtarzać dokładnie takiego samego tekstu z rozdziału „Funkcje". Jednak podobnie jak w przypadku funkcji, podstawową zasadą projektowania klas jest zachowanie małych rozmiarów. Podobnie jak w przypadku funkcji, naszym natychmiastowym pytaniem jest zawsze: „Jak małe?".

W przypadku funkcji mierzyliśmy ich rozmiar, zliczając linie fizyczne. W przypadku klas korzystamy z innej metryki. Zliczamy tzw. *odpowiedzialności*[1].

Na listingu 10.1 przedstawiony jest szkielet klasy `SuperDashboard`, która udostępnia 70 metod publicznych. Większość programistów zgodzi się, że jest ona zbyt duża. Niektórzy programiści mogą określać `SuperDashboard` jako „klasę boską".

LISTING 10.1. Zbyt wiele odpowiedzialności

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public String getCustomizerLanguagePath()
  public void setSystemConfigPath(String systemConfigPath)
  public String getSystemConfigDocument()
  public void setSystemConfigDocument(String systemConfigDocument)
  public boolean getGuruState()
  public boolean getNoviceState()
  public boolean getOpenSourceState()
  public void showObject(MetaObject object)
  public void showProgress(String s)
  public boolean isMetadataDirty()
  public void setIsMetadataDirty(boolean isMetadataDirty)
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public void setMouseSelectState(boolean isMouseSelected)
  public boolean isMouseSelected()
```

[1] [RDD].

```
  public LanguageManager getLanguageManager()
  public Project getProject()
  public Project getFirstProject()
  public Project getLastProject()
  public String getNewProjectName()
  public void setComponentSizes(Dimension dim)
  public String getCurrentDir()
  public void setCurrentDir(String newDir)
  public void updateStatus(int dotPos, int markPos)
  public Class[] getDataBaseClasses()
  public MetadataFeeder getMetadataFeeder()
  public void addProject(Project project)
  public boolean setCurrentProject(Project project)
  public boolean removeProject(Project project)
  public MetaProjectHeader getProgramMetadata()
  public void resetDashboard()
  public Project loadProject(String fileName, String projectName)
  public void setCanSaveMetadata(boolean canSave)
  public MetaObject getSelectedObject()
  public void deselectObjects()
  public void setProject(Project project)
  public void editorAction(String actionName, ActionEvent event)
  public void setMode(int mode)
  public FileManager getFileManager()
  public void setFileManager(FileManager fileManager)
  public ConfigManager getConfigManager()
  public void setConfigManager(ConfigManager configManager)
  public ClassLoader getClassLoader()
  public void setClassLoader(ClassLoader classLoader)
  public Properties getProps()
  public String getUserHome()
  public String getBaseDir()
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
  public MetaObject pasting(
    MetaObject target, MetaObject pasted, MetaProject project)
  public void processMenuItems(MetaObject metaObject)
  public void processMenuSeparators(MetaObject metaObject)
  public void processTabPages(MetaObject metaObject)
  public void processPlacement(MetaObject object)
  public void processCreateLayout(MetaObject object)
  public void updateDisplayLayer(MetaObject object, int layerIndex)
  public void propertyEditedRepaint(MetaObject object)
  public void processDeleteObject(MetaObject object)
  public boolean getAttachedToDesigner()
  public void processProjectChangedState(boolean hasProjectChanged)
  public void processObjectNameChanged(MetaObject object)
  public void runProject()
  public void setAçowDragging(boolean allowDragging)
  public boolean allowDragging()
  public boolean isCustomizing()
  public void setTitle(String title)
  public IdeMenuBar getIdeMenuBar()
  public void showHelper(MetaObject metaObject, String propertyName)
  // ... Wiele kolejnych niepublicznych metod ...
}
```

Co można powiedzieć w przypadku, gdy klasa `SuperDashboard` będzie zawierała wyłącznie metody z listingu 10.2?

LISTING 10.2. Wystarczająco mała?

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
}
```

Pięć metod to niezbyt dużo, prawda? W tym przypadku, pomimo małej liczby metod, `SuperDashboard` ma zbyt wiele *odpowiedzialności*.

Nazwa klasy powinna opisywać pełnione przez nią odpowiedzialności. W rzeczywistości nazewnictwo jest prawdopodobnie pierwszym sposobem na pomoc w określeniu wielkości klasy. Jeżeli nie możemy utworzyć spójnej nazwy klasy, prawdopodobnie jest ona zbyt duża. Im bardziej ogólna jest nazwa klasy, tym większe prawdopodobieństwo, że ma wiele odpowiedzialności. Na przykład nazwy klas zawierające takie słowa, jak `Processor`, `Manager` lub `Super`, często wskazują na niefortunną agregację odpowiedzialności.

Powinniśmy być w stanie napisać krótki opis klasy, używając około 25 słów, bez wykorzystania słów „jeżeli", „oraz", „lub" czy też „ale". Jak można opisać `SuperDashboard`? „Klasa `SuperDashboard` zapewnia dostęp do komponentu mającego ostatnio fokus oraz pozwala śledzić wersję i numer kompilacji". Słowo „oraz" jest podpowiedzią, że `SuperDashboard` ma zbyt dużo odpowiedzialności.

Zasada pojedynczej odpowiedzialności

Zasada pojedynczej odpowiedzialności (SRP)[2] zakłada, że klasa lub moduł powinny mieć jeden i tylko jeden *powód do zmiany*. Zasada ta zarówno definiuje odpowiedzialność, jak i daje wskazówki odnośnie do wielkości klas. Klasy powinny mieć jedną odpowiedzialność — jeden powód do zmiany.

Pozornie mała klasa `SuperDashboard` z listingu 10.2 ma dwa powody do zmiany. Po pierwsze, śledzi ona informacje o wersji, które będą prawdopodobnie aktualizowane przy każdym wydaniu oprogramowania. Po drugie, zarządza komponentami Java Swing (dziedziczy po `JFrame`, reprezentacji głównego okna GUI w bibliotece Swing). Pewne jest, że będziemy chcieli zmienić numer wersji po zmianie fragmentu kodu Swing, ale w odwrotnej sytuacji — niekoniecznie. Możemy zmieniać informacje o wersji w przypadku wprowadzania zmian w innych fragmentach kodu systemu.

Próba identyfikacji odpowiedzialności (powodów zmiany) często pomaga nam rozpoznać i utworzyć lepsze abstrakcje w naszym kodzie. Można bez trudu wydzielić trzy metody z klasy `SuperDashboard`, operujące na informacjach o wersji, do zewnętrznej klasy o nazwie `Version` (patrz listing 10.3). Klasa `Version` jest konstrukcją, która z dużym prawdopodobieństwem może być ponownie wykorzystywana w innych aplikacjach!

---

[2] Więcej na temat tej zasady można przeczytać w [PPP].

LISTING 10.3. Klasa o jednej odpowiedzialności

```
public class Version {
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
}
```

SRP jest jedną z najważniejszych koncepcji w projektowaniu obiektowym. Jest to również jedna z najprostszych koncepcji do zrozumienia oraz stosowania. Co jednak dziwne, jest jedną z najczęściej łamanych zasad projektowania. Regularnie spotykamy się z klasami, które wykonują zbyt dużo rzeczy. Dlaczego?

Doprowadzenie do tego, aby program działał, i tworzenie czystego oprogramowania to dwa bardzo różne zadania. Nasz umysł ma ograniczone możliwości, więc bardziej skupiamy się na doprowadzeniu kodu do stanu, w którym działa, niż na jego organizacji i czystości. Jest to całkowicie właściwe. Utrzymanie rozdziału zadań jest ważne zarówno dla *procesu* programowania, jak i dla naszych programów.

Problem wynika z tego, że wielu z nas uważa pracę za zakończoną po doprowadzeniu kodu do działania. Nie zajmujemy się *drugim* zadaniem — zapewnieniem organizacji i czystości kodu. Przechodzimy do kolejnego problemu, zamiast zajmować się dzieleniem przeciążonych klas na rozłączone jednostki z jedną odpowiedzialnością.

Jednocześnie wielu programistów boi się, że zbyt wiele małych i jednozadaniowych klas utrudni zrozumienie całego systemu. Obawiają się, że będą musieli przechodzić z klasy do klasy w celu sprawdzenia, w jaki sposób można wykonać większe zadanie.

Jednak system z wieloma małymi klasami nie ma więcej zmiennych części niż system z kilkoma dużymi klasami. System z kilkoma dużymi klasami wymaga dokładnie tyle samo nauki. Należy więc postawić pytanie: Czy wolimy przechowywać nasze narzędzia w skrzynkach z wieloma małymi szufladkami zawierającymi określone elementy, czy wolimy kilka szuflad, do których po prostu wrzucamy narzędzia?

Każdy większy system będzie zawierał dużą ilość kodu logiki i będzie skomplikowany. Podstawowym celem zarządzania tą złożonością jest jej *organizacja*, aby programista wiedział, gdzie szukać określonych elementów, i w danym momencie musiał analizować jedynie elementy bezpośrednio związane z problemem. Dla porównania, system z większymi, wielozadaniowymi klasami zawsze zmusza nas do przedzierania się przez wiele elementów, których w danym momencie nie potrzebujemy.

Powtórzmy to jeszcze raz: wolimy, gdy nasze systemy składają się z wielu małych klas, a nie kilku dużych. Każda mała klasa hermetyzuje jedną odpowiedzialność, ma jeden powód zmiany i dla osiągnięcia oczekiwanego działania systemu współpracuje z niewielką liczbą innych klas.

Spójność

Klasy powinny mieć niewielką liczbę zmiennych instancyjnych. Każda z metod klasy powinna manipulować jedną lub kilkoma tymi zmiennymi. Zwykle im większą liczbą zmiennych manipuluje klasa, tym bardziej spójna z klasą jest ta metoda. Klasa, w której każda zmienna jest wykorzystywana w każdej metodzie, jest maksymalnie spójna.

Zwykle nie jest zalecane ani możliwe tworzenie takich maksymalnie spójnych klas; z drugiej strony, spójność powinna być wysoka. Gdy spójność jest wysoka, oznacza to, że metody i zmienne klasy są wzajemnie zależne i tworzą logiczną całość.

Jako przykład weźmy implementację klasy `Stack` z listingu 10.4. Jest to tylko bardzo spójna klasa. W przypadku jej trzech metod jedynie `size()` nie korzysta z obu zmiennych.

LISTING 10.4. Stack.java — spójna klasa

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

Strategia tworzenia małych funkcji oraz krótkich list parametrów czasami prowadzi do wykorzystywania zmiennych instancyjnych używanych przez podzbiory metod. W takim przypadku niemal zawsze oznacza to, że przynajmniej jedna nowa klasa chce się uwolnić z bieżącej klasy. Powinniśmy spróbować podzielić zmienne i metody na dwie lub więcej klas, aby nowe klasy były bardziej spójne.

Utrzymywanie spójności powoduje powstanie wielu małych klas

Sam akt podziału dużych funkcji na mniejsze powoduje zwiększenie liczby klas. Weźmy pod uwagę dużą funkcję z zadeklarowanymi w niej wieloma zmiennymi. Załóżmy, że chcemy wydzielić małą część tej funkcji do osobnej funkcji. Jednak kod, jaki chcemy wydzielić, korzysta z czterech zmiennych zadeklarowanych w funkcji. Czy musimy przekazać wszystkie te zmienne do nowej funkcji jako argumenty?

Nie musimy! Jeżeli zmienimy te cztery zmienne na zmienne instancyjne klasy, będziemy mogli wydzielić kod bez przekazywania *jakichkolwiek* zmiennych. Bardzo *łatwo* będzie podzielić funkcję na mniejsze części.

Niestety, oznacza to również, że nasze klasy utracą spójność, ponieważ będzie się w nich gromadziło coraz więcej zmiennych, które istnieją tylko po to, aby kilka funkcji wspólnie z nich korzystało. Czekaj, czekaj! Jeżeli istnieje niewiele funkcji, które współużytkują określone zmienne, czy nie należy utworzyć z nich osobnej klasy? Oczywiście, że tak. Gdy klasy tracą spójność, należy je podzielić!

Tak więc podział dużej funkcji na mniejsze często daje nam również możliwość wydzielenia kilku mniejszych klas. Pozwala to na lepszą organizację programu i zapewnienie bardziej przejrzystej struktury.

Aby przedstawić, o co mi chodzi, skorzystam z od dawna znanego przykładu zamieszczonego w doskonałej książce Knutha, *Literate Programming*[3]. Na listingu 10.5 przedstawione jest tłumaczenie na język Java programu Knutha `PrintPrimes`. Aby być uczciwym wobec Knutha, muszę nadmienić, że program ten nie został przez niego napisany, ale jest wynikiem działania jego narzędzia WWW. Korzystam z niego, ponieważ jest to doskonały materiał do podziału dużej funkcji na mniejsze funkcje i klasy.

LISTING 10.5. PrintPrimes.java

```
package literatePrimes;

public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];

    J = 1;
    K = 1;
    P[1] = 2;
    ORD = 2;
    SQUARE = 9;

    while (K < M) {
      do {
        J = J + 2;
        if (J == SQUARE) {
```

[3] [Knuth92].

```
            ORD = ORD + 1;
            SQUARE = P[ORD] * P[ORD];
            MULT[ORD - 1] = J;
          }
          N = 2;
          JPRIME = true;
          while (N < ORD && JPRIME) {
            while (MULT[N] < J)
              MULT[N] = MULT[N] + P[N] + P[N];
            if (MULT[N] == J)
              JPRIME = false;
            N = N + 1;
          }
        } while (!JPRIME);
        K = K + 1;
        P[K] = J;
      }
      {
        PAGENUMBER = 1;
        PAGEOFFSET = 1;
        while (PAGEOFFSET <= M) {
          System.out.println("Pierwsze " + M +
                             " liczb pierwszych --- strona " + PAGENUMBER);
          System.out.println("");
          for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++){
            for (C = 0; C < CC;C++)
              if (ROWOFFSET + C * RR <= M)
                System.out.format("%10d", P[ROWOFFSET + C * RR]);
            System.out.println("");
          }
          System.out.println("\f");
          PAGENUMBER = PAGENUMBER + 1;
          PAGEOFFSET = PAGEOFFSET + RR * CC;
        }
      }
    }
  }
```

Program ten, zapisany w jednej funkcji, jest skomplikowany. Ma ściśle powiązaną, głęboko zagnieżdżoną strukturę i sporo dziwnych zmiennych. Co najwyżej można podzielić jedną dużą funkcję na kilka mniejszych.

Na listingach od 10.6 do 10.8 przedstawiony jest wynik podziału kodu z listingu 10.5 na mniejsze klasy i funkcje, jak również wynik wyboru znaczących nazw dla tych klas, funkcji i zmiennych.

LISTING 10.6. PrimePrinter.java (przebudowany)

```
package literatePrimes;

public class PrimePrinter {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

    final int ROWS_PER_PAGE = 50;
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
                               COLUMNS_PER_PAGE,
                               "Pierwsze " + NUMBER_OF_PRIMES +
                               " liczb pierwszych");
```

```
    tablePrinter.print(primes);
  }
}
```

LISTING 10.7. RowColumnPagePrinter.java

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;

  public RowColumnPagePrinter(int rowsPerPage,
                              int columnsPerPage,
                              String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
    numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
  }

  public void print(int data[]) {
    int pageNumber = 1;
    for (int firstIndexOnPage = 0;
         firstIndexOnPage < data.length;
         firstIndexOnPage += numbersPerPage) {
      int lastIndexOnPage =
        Math.min(firstIndexOnPage + numbersPerPage - 1,
                 data.length - 1);
      printPageHeader(pageHeader, pageNumber);
      printPage(firstIndexOnPage, lastIndexOnPage, data);
      printStream.println("\f");
      pageNumber++;
    }
  }

  private void printPage(int firstIndexOnPage,
                         int lastIndexOnPage,
                         int[] data) {
    int firstIndexOfLastRowOnPage =
      firstIndexOnPage + rowsPerPage - 1;
    for (int firstIndexInRow = firstIndexOnPage;
         firstIndexInRow <= firstIndexOfLastRowOnPage;
         firstIndexInRow++) {
      printRow(firstIndexInRow, lastIndexOnPage, data);
      printStream.println("");
    }
  }

  private void printRow(int firstIndexInRow,
                        int lastIndexOnPage,
                        int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
      int index = firstIndexInRow + column * rowsPerPage;
      if (index <= lastIndexOnPage)
        printStream.format("%10d", data[index]);
    }
```

```
    }

    private void printPageHeader(String pageHeader,
                                 int pageNumber) {
      printStream.println(pageHeader + " --- Strona " + pageNumber);
      printStream.println("");
    }

    public void setOutput(PrintStream printStream) {
      this.printStream = printStream;
    }
  }
```

LISTING 10.8. PrimeGenerator.java

```
  package literatePrimes;

  import java.util.ArrayList;

  public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;

    protected static int[] generate(int n) {
      primes = new int[n];
      multiplesOfPrimeFactors = new ArrayList<Integer>();
      set2AsFirstPrime();
      checkOddNumbersForSubsequentPrimes();
      return primes;
    }

    private static void set2AsFirstPrime() {
      primes[0] = 2;
      multiplesOfPrimeFactors.add(2);
    }

    private static void checkOddNumbersForSubsequentPrimes() {
      int primeIndex = 1;
      for (int candidate = 3;
           primeIndex < primes.length;
           candidate += 2) {
        if (isPrime(candidate))
          primes[primeIndex++] = candidate;
      }
    }

    private static boolean isPrime(int candidate) {
      if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
        multiplesOfPrimeFactors.add(candidate);
        return false;
      }
      return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
    }

    private static boolean
    isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
      int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
      int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
      return candidate == leastRelevantMultiple;
    }

    private static boolean
    isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
```

```
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
      if (isMultipleOfNthPrimeFactor(candidate, n))
        return false;
    }
    return true;
  }

  private static boolean
  isMultipleOfNthPrimeFactor(int candidate, int n) {
    return
      candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
  }

  private static int
  smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
      multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
  }
}
```

Pierwszą rzeczą, jaką zauważamy, jest fakt, że program stał się znacznie dłuższy. Wzrósł z niewiele ponad jednej strony do niemal trzech. Istnieje kilka przyczyn tego wzrostu objętości kodu. Po pierwsze, przebudowany program korzysta z dłuższych, bardziej opisowych nazw zmiennych. Po drugie, przebudowany program korzysta z deklaracji funkcji i klas jako sposobu dodawania komentarzy do kodu. Po trzecie, wykorzystaliśmy odstępy i techniki formatowania pozwalające zachować czytelność programu.

Warto zauważyć, że program został podzielony na trzy główne odpowiedzialności. Główny program znajduje się w samej klasie `PrimePrinter`. Jej odpowiedzialnością jest obsługa środowiska wykonania. Zmienia się ona, gdy zmieniają się wywołania metod. Jeżeli program ten zostałby na przykład skonwertowany na usługę SOAP, zmieniona zostałaby ta klasa.

Klasa `RowColumnPagePrinter` obsługuje wszystkie operacje formatowania list liczb na stronach o określonej liczbie wierszy i kolumn. Jeżeli zmiany będzie wymagało formatowanie danych wyjściowych, zostanie zmodyfikowana ta klasa.

Klasa `PrimeGenerator` obsługuje wszystkie operacje generowania listy liczb pierwszych. Należy zauważyć, że nie jest przewidziane tworzenie tego obiektu. Klasa jest po prostu użytecznym zakresem, w którym można zadeklarować zmienne i zapewnić ich ukrycie. Klasa ta zostanie zmieniona, jeżeli zmieni się algorytm obliczania liczb pierwszych.

To nie było ponowne napisanie programu! Nie zaczęliśmy od pustego ekranu i nie napisaliśmy programu od podstaw. W rzeczywistości, jeżeli przyjrzymy się dokładniej tym dwóm programom, okaże się, że korzystają z tego samego algorytmu i tych samych mechanizmów.

Zmianą, jaką wprowadziliśmy, było dodanie zestawu testów *precyzyjnie* weryfikujących działanie pierwszego programu. Następnie wprowadzono wiele małych zmian, jedna po drugiej. Po każdej zmianie program był uruchamiany w celu upewnienia się, że jego działanie nie zmieniło się.

Organizowanie zmian

W większości systemów jedynym stałym elementem są zmiany. Każda zmiana powoduje zagrożenie, że pozostała część systemu przestanie działać w zaplanowany sposób. W czystym systemie organizujemy swoje klasy w taki sposób, aby zredukować ryzyko zmian.

Klasa `Sql` z listingu 10.9 jest wykorzystywana do generowania prawidłowo sformatowanego zapytania SQL na podstawie metadanych. Jest ona w trakcie rozwoju i między innymi nie obsługuje operacji SQL `update`. Gdy przyszedł czas na dodanie do klasy `Sql` obsługi zapytania `update`, musieliśmy „otworzyć" tę klasę, aby wprowadzić modyfikacje. Z otwieraniem klasy zawsze związane jest pewne ryzyko. Każda modyfikacja klasy może spowodować błąd w innym fragmencie kodu klasy. Musi ona zostać w pełni przetestowana.

LISTING 10.9. Klasa, która musi być otwarta na zmiany

```
public class Sql {
  public Sql(String table, Column[] columns)
  public String create()
  public String insert(Object[] fields)
  public String selectAll()
  public String findByKey(String keyColumn, String keyValue)
  public String select(Column column, String pattern)
  public String select(Criteria criteria)
  public String preparedInsert()
  private String columnList(Column[] columns)
  private String valuesList(Object[] fields, final Column[] columns)
  private String selectWithCriteria(String criteria)
  private String placeholderList(Column[] columns)
}
```

Gdy dodajemy nowy rodzaj wyrażenia, klasa `Sql` musi zostać zmieniona. Musi ulec zmianie również w momencie, gdy zmienimy szczegóły jednego typu wyrażenia — na przykład jeżeli musimy zmodyfikować zapytanie `select`, aby obsługiwało podzapytania. Dwa powody zmiany oznaczają, że klasa `Sql` narusza zasadę SRP.

Możemy zauważyć to naruszenie SRP na podstawie prostego organizacyjnego punktu widzenia. Metoda przedstawiona dla `Sql` pokazuje, że istnieją metody prywatne, takie jak `selectWithCriteria`, które odnoszą się tylko do instrukcji `select`.

Metody prywatne, które odnoszą się tylko do małej części klasy, mogą być przydatne do lokalizowania potencjalnych obszarów usprawnienia. Jednak podstawową zachętą do podjęcia akcji powinna być sama zmiana systemu. Jeżeli klasa `Sql` jest uznana za logicznie kompletną, nie ma konieczności rozdzielania odpowiedzialności. Jeżeli nie potrzebujemy funkcji `update` w dającej się przewidzieć przyszłości, powinniśmy zostawić klasę `Sql`. Jednak gdy zdecydujemy się otworzyć klasę, powinniśmy rozważyć poprawienie naszego projektu.

Przyjmijmy rozwiązanie takie jak na listingu 10.10. Każda metoda interfejsu publicznego, zdefiniowana w poprzedniej klasie `Sql` z listingu 10.9, została przeniesiona do osobnej klasy dziedziczącej po klasie `Sql`. Metody prywatne, takie jak `valuesList`, zostały przeniesione bezpośrednio tam, gdzie są potrzebne. Wspólne operacje prywatne zostały wydzielone do pary klas narzędziowych, `Where` oraz `ColumnList`.

LISTING 10.10. Zbiór zamkniętych klas

```
abstract public class Sql {
  public Sql(String table, Column[] columns)
  abstract public String generate();
}

public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}

public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}

public class InsertSql extends Sql {
  public InsertSql(String table, Column[] columns, Object[] fields)
  @Override public String generate()
  private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
  public SelectWithCriteriaSql(
  String table, Column[] columns, Criteria criteria)
  @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
  public SelectWithMatchSql(
  String table, Column[] columns, Column column, String pattern)
  @Override public String generate()
}

public class FindByKeySql extends Sql
  public FindByKeySql(
  String table, Column[] columns, String keyColumn, String keyValue)
  @Override public String generate()
}

public class PreparedInsertSql extends Sql {
  public PreparedInsertSql(String table, Column[] columns)
  @Override public String generate() {
  private String placeholderList(Column[] columns)
}

public class Where {
  public Where(String criteria)
  public String generate()
}

public class ColumnList {
  public ColumnList(Column[] columns)
  public String generate()
}
```

Kod w każdej klasie stał się niezwykle prosty. Wymagany czas skupienia w celu zrozumienia każdej z klas skurczył się niemal do zera. Ryzyko, że jedna funkcja spowoduje błąd w innej, wyraźnie zmalało. Z punktu widzenia testów znacznie prostszym zadaniem stało się sprawdzenie każdego fragmentu tego rozwiązania, ponieważ klasy są izolowane od siebie.

Równie ważne jest to, że gdy przyjdzie czas na dodanie instrukcji `update`, żadna z istniejących klas nie będzie wymagała zmiany! Kod pozwalający na budowanie instrukcji `update` umieścimy w nowej klasie dziedziczącej po `Sql`, o nazwie `UpdateSql`. Żaden inny kod systemu nie ulegnie uszkodzeniu z powodu wprowadzenia tej zmiany.

Nasza zrestrukturyzowana logika klasy `Sql` łączy najlepsze cechy obu światów. Wspiera ona SRP. Wspiera również inną ważną zasadę projektowania obiektowego, zasadę otwarty-zamknięty, czyli OCP[4]. Klasy powinny być otwarte na rozszerzanie, ale zamknięte dla modyfikacji. Nasza zmieniona klasa `Sql` jest otwarta na dodawanie nowych funkcji przez dziedziczenie, a zmiany te możemy wykonywać bez konieczności otwierania pozostałych klas. Po prostu dorzucamy naszą klasę `UpdateSql` do pakietu.

Powinniśmy tak konstruować systemy, aby przy ich aktualizowaniu o nowe lub zmienione funkcje nie było konieczne wprowadzanie zbyt wielu dodatkowych modyfikacji. W idealnym systemie wprowadzamy nowe funkcje przez jego rozszerzanie, a nie wprowadzanie zmian do istniejącego kodu.

Izolowanie modułów kodu przed zmianami

Potrzeby wymuszają zmiany, dlatego kod podlega modyfikacji. Podstawy programowania obiektowego mówią, że występują klasy konkretne, zawierające szczegóły implementacji (kod), oraz klasy abstrakcyjne, które reprezentują wyłącznie koncepcje. W przypadku gdy klasa klienta zależy od szczegółów klasy konkretnej, istnieje ryzyko, że szczegóły te mogą ulec zmianie. Aby zabezpieczyć się przed wpływem tych szczegółów, wprowadzamy interfejsy i klasy abstrakcyjne.

Zależności od szczegółów klas konkretnych powodują problemy z testowaniem systemu. Jeżeli budujemy klasę `Portfolio`, która zależy od zewnętrznego API `TokyoStockExchange`, do dziedziczenia wartości portfela, nasze przypadki testowe są obciążone zmiennością tego wyszukiwania. Trudno napisać test, jeżeli co pięć minut otrzymujemy inną odpowiedź!

Zamiast projektowania klasy `Portfolio`, która bezpośrednio zależy od `TokyoStockExchange`, powinniśmy utworzyć interfejs `StockExchange`, który zawiera jedną metodę:

```
public interface StockExchange {
  Money currentPrice(String symbol);
}
```

Klasę `TokyoStockExchange` projektujemy tak, aby implementowała ten interfejs. Musimy się również upewnić, że konstruktor `Portfolio` jako argumentu oczekuje referencji do `StockExchange`:

```
public Portfolio {
  private StockExchange exchange;
  public Portfolio(StockExchange exchange) {
    this.exchange = exchange;
  }
  // ...
}
```

[4] [PPP].

Teraz możemy utworzyć testową implementację interfejsu `StockExchange`, która emuluje `TokyoStockExchange`. Taka testowa implementacja zwraca stałą wartość dla dowolnego przekazanego symbolu akcji. Jeżeli nasz test demonstruje zakup pięciu akcji firmy Microsoft do naszego portfela, możemy użyć implementacji testowej zwracającej zawsze wartość 100 dolarów za akcję. Nasza testowa implementacja interfejsu `StockExchange` sprowadza się do zwykłego wyszukania w tablicy. Możemy teraz napisać test, który oczekuje, aby całkowita wartość portfela wynosiła 500 dolarów.

```
public class PortfolioTest {
  private FixedStockExchangeStub exchange;
  private Portfolio portfolio;

  @Before
  protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
  }

  @Test
  public void GivenFiveMSFTTotalShouldBe500() throws Exception {
    portfolio.add(5, "MSFT");
    Assert.assertEquals(500, portfolio.value());
  }
}
```

Jeżeli moduły systemu są wystarczająco niezależne, aby można było je testować w ten sposób, to cały system jest również bardziej elastyczny i ułatwia ponowne wykorzystanie kodu. Brak sprzężeń oznacza, że elementy systemu są lepiej izolowane od siebie oraz od zmian. Izolacja ta ułatwia zrozumienie każdego elementu systemu.

Przez minimalizację sprzężeń nasze klasy są zgodne z inną zasadą projektowania klas, znaną jako zasada odwrócenia zależności (DIP)[5]. Zasada ta głosi, że klasy powinny zależeć od abstrakcji, a nie klas konkretnych.

Nasza klasa `Portfolio` nie zależy już od szczegółów implementacji klasy `TokyoStockExchange`, ale od interfejsu `StockExchange`. Interfejs `StockExchange` reprezentuje abstrakcyjną koncepcję odpytania o bieżącą cenę akcji. Abstrakcja ta izoluje wszystkie szczegóły uzyskiwania tej ceny, w tym również tego, skąd jest uzyskiwana cena.

Bibliografia

[RDD]: Rebecca Wirfs-Brock i inni, *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley 2002.

[PPP]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall 2002.

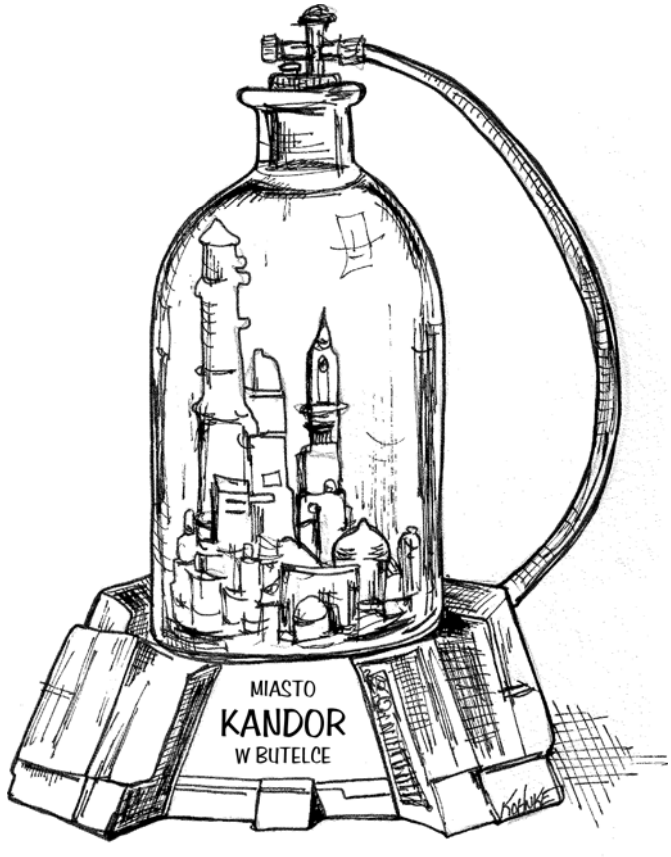
[Knuth92]: Donald E. Knuth, *Literate Programming*, Center for the Study of language and Information, Leland Stanford Junior University 1992.

[5] [PPP].

ROZDZIAŁ 11.

Systemy

Dr Kevin Dean Wampler



Złożoność zabija. Zatruwa życie programistów i utrudnia planowanie, budowanie i testowanie produktów.

Ray Ozzie, CTO, Microsoft Corporation

Jak budowałbyś miasto?

Czy można samodzielnie zarządzać wszystkimi szczegółami jakiegoś dużego systemu? Prawdopodobnie nie. Nawet zarządzanie istniejącym miastem wykracza poza możliwości jednej osoby. Jednak miasta funkcjonują (w większości przypadków). Funkcjonują, ponieważ istnieją zespoły ludzi, którzy zarządzają określonymi częściami miasta, systemem wodociągowym, systemem zasilania, sterowania ruchem, pilnowaniem prawa, numerami budynków i tak dalej. Część z tych ludzi jest odpowiedzialna za *ogólną koncepcję*, natomiast inni skupiają się na szczegółach.

Miasta funkcjonują, ponieważ wypracowane zostały odpowiednie poziomy abstrakcji i modularności, które pozwalają poszczególnym osobom i zarządzanym przez nie „komponentom" pracować efektywnie, bez konieczności rozumienia działania wszystkich mechanizmów.

Choć zespoły programistyczne są również podobnie zorganizowane, systemy, nad którymi pracują, często nie mają podobnej separacji zadań i poziomów abstrakcji. Czysty kod pozwala nam osiągnąć to na niższych poziomach abstrakcji. W tym rozdziale zajmiemy się sposobami zachowania czystości na wyższym poziomie abstrakcji — na poziomie *systemu*.

Oddzielenie konstruowania systemu od jego używania

Na początek trzeba zauważyć, że *konstruowanie* i *używanie* to zupełnie inne procesy. W czasie, gdy piszę te słowa, z mojego okna w Chicago widzę budowę nowego hotelu. Obecnie jest to nagie cementowe pudło z żurawiami konstrukcyjnymi oraz windami przyczepionymi na jego ścianach. Wszyscy pracujący tam ludzie noszą kaski i ubrania robocze. Po mniej więcej roku hotel zostanie ukończony. Żurawie i winda znikną. Budynek będzie czysty, obudowany atrakcyjną wizualnie szklaną elewacją. Pracownicy i goście hotelu również będą wyglądali inaczej.

> *Systemy oprogramowania powinny oddzielić proces uruchamiania, w którym są budowane obiekty aplikacji i łączone ze sobą zależności, od logiki działania, która jest wykorzystywana po uruchomieniu.*

Proces uruchomienia jest *problemem*, który musi być rozwiązany w każdej aplikacji. Jest to pierwszy problem, jaki przeanalizujemy w tym rozdziale. Jedną z najstarszych i najważniejszych technik projektowania jest *rozdzielenie problemów*.

Niestety, w większości aplikacji problem ten nie jest wydzielony. Kod dla procesu uruchamiania jest tworzony ad hoc i jest wymieszany z logiką działania. Poniżej przedstawiony jest typowy przykład:

```
public Service getService() {
  if (service == null)
    service = new MyServiceImpl(...); // Odpowiednie wartości domyślne dla większości przypadków?
  return service;
}
```

Jest to idiom późnej inicjalizacji (wartościowania), który ma kilka zalet. Nie ponosimy kosztu utworzenia, jeżeli aktualnie korzystamy z takiego obiektu, a w wyniku zastosowania tego mechanizmu czas uruchomienia może być mniejszy. Zapewnia on również, że nigdy nie zostanie zwrócona wartość `null`.

Jednak powoduje to wbudowanie zależności od `MyServiceImpl` i wszystkich elementów wymaganych przez konstruktor (które tu pominąłem). Nie możemy skompilować kodu bez rozwiązania tych zależności, nawet jeżeli w rzeczywistości nie korzystamy w czasie jego działania z obiektów tego typu!

Problemem może być testowanie. Jeżeli `MyServiceImpl` jest ciężkim obiektem, musimy się upewnić, że przed wywołaniem metod w czasie testowania jednostkowego do pola usługi przypisane zostały odpowiednie obiekty *Test Double*[1] lub *Mock*. Ponieważ kod konstrukcyjny jest wymieszany z normalnym przetwarzaniem, powinniśmy przetestować wszystkie ścieżki wykonania (na przykład test `null` i jego bloku). Posiadanie obu tych odpowiedzialności oznacza, że metoda wykonuje więcej niż jedną operację, więc łamie ona *zasadę pojedynczej odpowiedzialności*.

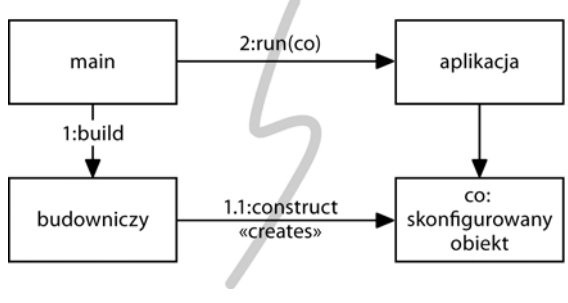
Jednak chyba gorsze dla nas jest to, że nie wiemy, czy `MyServiceImpl` jest właściwym obiektem dla wszystkich przypadków. Zapisałem to w komentarzu. Dlaczego klasa zawierająca tę metodę musi znać kontekst globalny? Czy *kiedykolwiek* będziemy wiedzieć, jaki jest właściwy obiekt do wykorzystania w tym miejscu? Czy jest możliwe, że jeden typ będzie odpowiedni dla wszystkich możliwych kontekstów?

Jedno wystąpienie późnej inicjalizacji nie jest oczywiście poważnym problemem. Jednak zwykle w aplikacji tego typu idiom występuje wiele razy. Dlatego globalna *strategia* konfiguracji (jeżeli istnieje) jest *rozsiana* w całej aplikacji, zachowując niewielką modularność, i często jest wielokrotnie powielana.

Jeżeli *aktywnie* budujemy odpowiednio skonstruowane i solidne systemy, nie powinniśmy pozwalać niewielkim, *wygodnym* idiomom na złamanie modularności. Proces tworzenia obiektów i jego konfiguracji nie jest wyjątkiem. Powinniśmy zmodularyzować ten proces, oddzielając go od normalnej logiki działania, oraz upewnić się, że mamy globalną, spójną strategię rozwiązywania głównych zależności.

Wydzielenie modułu main

Jednym ze sposobów oddzielenia *konstrukcji* od *używania* jest proste przeniesienie wszystkich aspektów tworzenia do `main` lub modułów wywoływanych przez `main` oraz zaprojektowanie pozostałych części systemu tak, aby zakładały, że wszystkie obiekty są odpowiednio utworzone i skonfigurowane (patrz rysunek 11.1).



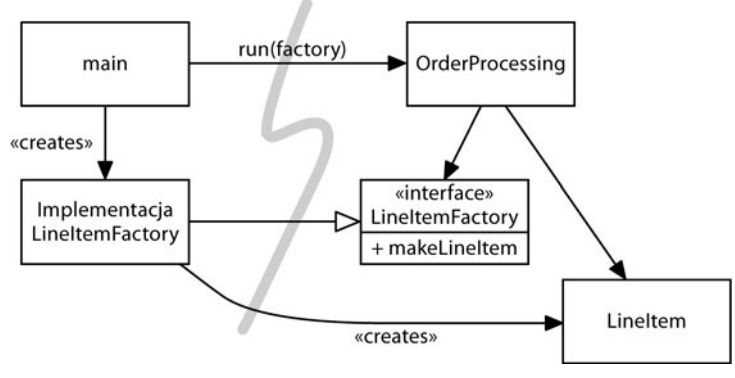
RYSUNEK 11.1. Wydzielenie konstrukcji do main()

[1] [Mezzaros07].

Przebieg sterowania jest łatwy w śledzeniu. Funkcja `main` buduje wszystkie niezbędne obiekty systemu, a następnie przekazuje je do aplikacji, która po prostu z nich korzysta. Należy zwrócić uwagę na kierunek strzałek zależności przechodzących przez barierę pomiędzy `main` i aplikacją. Wszystkie wskazują w jednym kierunku, od `main` na zewnątrz. Oznacza to, że aplikacja nie ma żadnych informacji o funkcji `main` ani o procesie tworzenia. Po prostu oczekuje, że wszystko zostało prawidłowo zbudowane.

Fabryki

Czasami konieczne jest, aby aplikacja była odpowiedzialna za to, *kiedy* zostanie utworzony obiekt. Na przykład w systemie przetwarzania zamówień aplikacja musi utworzyć obiekty `LineItem` w celu dodania ich do `Order`. W takim przypadku możemy skorzystać ze wzorca fabryki abstrakcyjnej[2] (ang. *Abstract Factory*) w celu przekazania aplikacji kontroli nad tym, kiedy należy zbudować obiekty `LineItem`, ale szczegóły jego tworzenia pozostaną oddzielone od kodu aplikacji (patrz rysunek 11.2).



RYSUNEK 11.2. Oddzielenie konstrukcji za pomocą fabryki abstrakcyjnej

I w tym przypadku powinniśmy zauważyć, że wszystkie zależności są skierowane z `main` do aplikacji `OrderProcessing`. Oznacza to, że aplikacja jest oddzielona od szczegółów tworzenia obiektu `LineItem`. Funkcja ta jest dostępna w `LineItemFactoryImplementation`, która znajduje się na rysunku po stronie `main`. Dzięki temu aplikacja ma pełną kontrolę nad momentem utworzenia obiektu `LineItem`, a nawet może przekazywać do konstruktora argumenty specyficzne dla aplikacji.

Wstrzykiwanie zależności

Zaawansowanym mechanizmem oddzielania konstrukcji od użytkowania jest *wstrzykiwanie zależności* (ang. *Dependency Injection* — DI), będące zastosowaniem *odwrócenia sterowania* (ang. *Inversion of Control* — IoC) do zarządzania zależnościami[3]. Odwrócenie sterowania przenosi drugorzędne odpowiedzialności z obiektu do innego obiektu, dedykowanego do tych zadań, przez co zapewnia się zachowanie *zasady pojedynczej odpowiedzialności*. W kontekście zarządzania zależnościami

[2] [GOF].

[3] Przykłady można znaleźć w [Fowler].

obiekt nie powinien być odpowiedzialny za samodzielne tworzenie zależności. Powinien przekazać tę odpowiedzialność do innego „autorytarnego" mechanizmu, odwracając w ten sposób sterowanie. Ponieważ konfiguracja jest problemem globalnym, ten autorytarny mechanizm zwykle będzie modułem „main" lub specjalizowanym *kontenerem*.

Wyszukiwanie JNDI jest „częściową" implementacją DI, w której obiekt odpytuje serwer katalogu w celu dostarczenia „usługi" odpowiadającej określonej nazwie.

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

Obiekt wywołujący nie steruje rodzajem zwracanego obiektu (o ile oczywiście implementuje on odpowiedni interfejs), ale wywołujący obiekt nadal aktywnie rozwiązuje zależności.

Prawdziwe wstrzykiwanie zależności idzie o krok dalej. Klasa nie wykonuje bezpośrednich kroków w celu rozwiązania jej zależności — jest całkowicie pasywna. Zamiast tego udostępnia metody ustawiające lub argumenty konstruktora (albo oba te mechanizmy), które są używane do *wstrzykiwania* zależności. W czasie procesu konstrukcji kontener DI tworzy wymagane obiekty (zwykle na żądanie) i korzysta z argumentów konstruktora lub metod ustawiających do dowiązania zależności. To, które zależne obiekty są w rzeczywistości użyte, jest definiowane w pliku konfiguracyjnym lub programowo w specjalizowanym module konstrukcyjnym.

Najlepszym znanym kontenerem DI dla języka Java jest Spring Framework[4]. W pliku XML definiujemy, które obiekty należy ze sobą połączyć, a następnie w kodzie Java pytamy o określone obiekty z użyciem ich nazwy. Odpowiedni przykład zostanie przedstawiony w dalszej części rozdziału.

Co jednak z zaletami późnej inicjalizacji? W przypadku DI idiom ten jest czasami przydatny. Po pierwsze, większość kontenerów DI nie tworzy obiektów do momentu, aż są potrzebne. Po drugie, wiele z tych kontenerów udostępnia mechanizmy wywoływania fabryk abstrakcyjnych lub pośredników konstrukcji, które mogą być użyte do późnego wartościowania lub podobnych *optymalizacji*[5].

Skalowanie w górę

Miasta wyrastają z miasteczek, które z kolei rozwijają się z osad. Na początku drogi są wąskie i niewygodne, następnie są utwardzane, a z czasem poszerzane. Małe budynki i puste przestrzenie są zastępowane większymi budynkami, których część w końcu zostanie zastąpiona drapaczami chmur.

Na początku niedostępne są takie usługi jak energia, woda, odprowadzanie ścieków, a nawet internet (ech!). Usługi te są dodawane wraz ze wzrostem populacji i gęstości budynków.

Wzrost ten nie przebiega bezboleśnie. Ile razy jechaliśmy zderzak w zderzak przez miejsce, w którym był prowadzony projekt „ulepszania" drogi, i pytaliśmy się: „Dlaczego nie zbudowano od razu drogi o odpowiedniej szerokości?".

---

[4] Patrz [Spring]. Istnieje również biblioteka Spring.NET.

[5] Nie należy zapominać, że późne tworzenie i wartościowanie jest tylko optymalizacją, i to prawdopodobnie przedwczesną!

Jednak nie może się to dziać w inny sposób. Kto uzasadni wydanie góry pieniędzy na sześciopasmową autostradę przez środek małego miasteczka, które prawdopodobnie kiedyś się powiększy? Kto *chciałby* mieć taką drogę w swoim miasteczku?

Mitem jest, że możemy otrzymać „właściwe systemy za pierwszym razem". Zamiast tego powinniśmy wdrażać tylko obecne *potrzeby*, a następnie przebudowywać i rozszerzać system, aby spełniał potrzeby w przyszłości. Jest to esencja wielokrotnej i iteracyjnej solidności. Programowanie sterowane testami, przebudowywanie oraz tworzenie czystego kodu pozwala na pracę na poziomie kodu.

Co jednak można powiedzieć o poziomie systemowym? Czy architektura systemu wymaga wstępnego planowania? Czy nie może ona zwiększać się przyrostowo od prostej do złożonej?

> *Systemy oprogramowania są inne niż fizyczne systemy. Ich architektury mogą się zwiększać przyrostowo,* ***jeżeli*** *zapewnimy odpowiednią separację problemów.*

Jak się okazuje, ulotna natura systemów programowych pozwala na takie planowanie. Na początek przedstawimy negatywny przykład architektury, w której problemy zostały nieodpowiednio odseparowane.

Oryginalne architektury EJB1 oraz EJB2 nie rozdzielały odpowiednio problemów i przez to wprowadzały niepotrzebne bariery wzrostu organicznego. Weźmy jako przykład *Entity Bean* dla trwałej klasy `Bank`. Entity Bean jest pamięciową reprezentacją danych relacyjnych, czyli, mówiąc inaczej, wiersza w tabeli.

Po pierwsze, konieczne było zdefiniowanie lokalnych (wewnątrz procesu) lub zdalnych (w osobnym środowisku JVM) interfejsów używanych przez klientów. Na listingu 11.1 przedstawiony jest przykładowy interfejs lokalny.

LISTING 11.1. Lokalny interfejs EJB2 dla EJB Bank

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
  String getStreetAddr1() throws EJBException;
  String getStreetAddr2() throws EJBException;
  String getCity() throws EJBException;
  String getState() throws EJBException;
  String getZipCode() throws EJBException;
  void setStreetAddr1(String street1) throws EJBException;
  void setStreetAddr2(String street2) throws EJBException;
  void setCity(String city) throws EJBException;
  void setState(String state) throws EJBException;
  void setZipCode(String zip) throws EJBException;
  Collection getAccounts() throws EJBException;
  void setAccounts(Collection accounts) throws EJBException;
  void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Przedstawionych jest tu kilka atrybutów adresu banku oraz kolekcja kont prowadzonych przez bank, z których każde będzie posiadało własne dane obsługiwane przez osobny EJB `Account`. Na listingu 11.2 przedstawiona jest odpowiednia implementacja klasy `Bank`.

LISTING 11.2. Implementacja Entity Bean w EJB2

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
  // Logika biznesowa...
  public abstract String getStreetAddr1();
  public abstract String getStreetAddr2();
  public abstract String getCity();
  public abstract String getState();
  public abstract String getZipCode();
  public abstract void setStreetAddr1(String street1);
  public abstract void setStreetAddr2(String street2);
  public abstract void setCity(String city);
  public abstract void setState(String state);
  public abstract void setZipCode(String zip);
  public abstract Collection getAccounts();
  public abstract void setAccounts(Collection accounts);
  public void addAccount(AccountDTO accountDTO) {
    InitialContext context = new InitialContext();
    AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
    AccountLocal account = accountHome.create(accountDTO);
    Collection accounts = getAccounts();
    accounts.add(account);
  }
  // Logika kontenera EJB.
  public abstract void setId(Integer id);
  public abstract Integer getId();
  public Integer ejbCreate(Integer id) { ... }
  public void ejbPostCreate(Integer id) { ... }
  // Pozostała część musi być zaimplementowana, ale z reguły metody są puste:
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void ejbRemove() {}
}
```

Nie zamieszczamy tu odpowiedniego interfejsu *LocalHome*, czyli fabryki wykorzystywanej do tworzenia obiektów, ani możliwych metod wyszukujących dla klasy `Bank`.

Na koniec musimy napisać w XML-u jeden lub więcej deskryptorów instalacyjnych, które definiują szczegóły mapowania obiektowo-relacyjnego dla magazynu, oczekiwane działanie transakcji, ograniczenia bezpieczeństwa i tak dalej.

Logika biznesowa jest ściśle złączona z „kontenerem” aplikacji EJB2. Konieczne jest dziedziczenie po typach kontenera i trzeba napisać wiele metod cyklu życia, wymaganych przez kontener.

Z powodu tego sprzężenia z ciężkim kontenerem wyizolowanie testów jednostkowych jest trudne. Konieczne jest napisanie imitacji kontenera, co jest trudne, lub marnowanie wiele czasu na instalowanie EJB i wykonywanie testów na rzeczywistym serwerze. Ponowne użycie kodu poza architekturą EJB2 jest niemożliwe z powodu tego ścisłego sprzężenia.

Nawet programowanie obiektowe jest ograniczone. Jeden bean nie może dziedziczyć po innym. Zwróćmy uwagę na kod dodawania nowego konta. Standardem w EJB2 jest definiowanie „obiektów transferu danych" (DTO), które są w rzeczywistości strukturami bez żadnych operacji. Zwykle prowadzi to do powstania nadmiarowych typów przechowujących dokładnie te same dane i wymaga kodu zapewniającego kopiowanie danych z jednego obiektu do drugiego.

Separowanie (rozcięcie) problemów

Architektura EJB2 w niektórych obszarach zbliża się do separacji problemów. Na przykład wymagane działanie transakcji, zabezpieczeń i niektóre zachowania mechanizmów trwałości są deklarowane w deskryptorze instalacji, niezależnie od kodu źródłowego.

Należy zauważyć, że *problemy* takie jak trwałość obiektów zwykle mają tendencję do przechodzenia przez naturalne granice obiektów domeny. Zazwyczaj chcemy zapisać wszystkie obiekty z użyciem tej samej strategii, na przykład korzystając z określonego systemu DBMS[6] zamiast płaskich plików, korzystając z określonej strategii nazewnictwa dla tabeli i kolumn, z użyciem spójnej semantyki obiektowej i tak dalej.

W teorii można opisywać naszą strategię trwałości w modularny, hermetyzowany sposób. Jednak w praktyce musimy powielić właściwie ten sam kod, implementujący strategię trwałości, w wielu obiektach. Do tego typu zagadnień stosujemy określenie *rozcięcie problemów*. I w tym przypadku biblioteka trwałości może być modularna, jak również modularna może być nasza logika domeny, traktowana osobno. Problemy wynikają z *przecięcia* tych domen.

W rzeczywistości sposób, w jaki architektura EJB obsługuje trwałość obiektów, bezpieczeństwo aplikacji i transakcje, „przewiduje" *programowanie zorientowane aspektowo* (AOP)[7], które jest ogólnym podejściem do przywracania modularności w przypadku rozcięcia problemów.

W AOP konstrukcje modularne nazywane *aspektami* definiują te punkty systemu, których działanie należy zmienić w pewien spójny sposób, tak by obsługiwały określone zagadnienie. Specyfikacja ta jest wykonywana z użyciem zwięzłego mechanizmu deklaratywnego lub programowego.

W przypadku mechanizmu trwałości obiektów możemy zadeklarować, które obiekty i atrybuty (lub ich *wzorce*) powinny być utrwalane, a następnie wydelegować zadanie utrwalania do odpowiedniej biblioteki. Zadanie modyfikowania jest realizowane przez bibliotekę AOP *nieinwazyjnie*[8] dla docelowego kodu. Zapoznamy się teraz z trzema mechanizmami aspektowymi lub podobnymi do aspektowych w języku Java.

---

[6] Systemy zarządzania bazą danych.

[7] Więcej informacji na temat aspektów można znaleźć w [AOSD], a informacje na temat Aspect-J w [AspectJ] i [Colyer].

[8] Czyli nie ma potrzeby edycji docelowego kodu źródłowego.

Pośredniki Java

Mechanizm pośredników Java jest odpowiedni dla prostych przypadków, takich jak otaczanie wywołań metody w pojedynczych obiektach lub klasach. Jednak dynamiczne pośredniki zapewniane w JDK operują tylko na interfejsach. Aby zapewnić pośredniczenie dla klas, należy skorzystać z bibliotek manipulujących kodem wynikowym, takim jak CGLIB, ASM lub Javassist[9].

Na listingu 11.3 przedstawiony jest szkielet pośrednika Java zapewniającego obsługę trwałości dla naszej aplikacji `Bank`, obsługujący tylko metody pobierania i ustawiania list kont.

LISTING 11.3. Przykład pośrednika JDK

```
// Bank.java (pominięte nazwy pakietów...).
import java.utils.*;

// Abstrakcja banku.
public interface Bank {
  Collection<Account> getAccounts();
  void setAccounts(Collection<Account> accounts);
}

// BankImpl.java
import java.utils.*;

// Obiekt "Plain Old Java Object" (POJO) implementujący abstrakcję.
public class BankImpl implements Bank {
  private List<Account> accounts;

  public Collection<Account> getAccounts() {
    return accounts;
  }
  public void setAccounts(Collection<Account> accounts) {
    this.accounts = new ArrayList<Account>();
    for (Account account: accounts) {
      this.accounts.add(account);
    }
  }
}

// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;
// "InvocationHandler" wymagany przez API pośrednika.
public class BankProxyHandler implements InvocationHandler {
  private Bank bank;

  public BankHandler (Bank bank) {
    this.bank = bank;
  }

  // Metoda zdefiniowana w InvocationHandler
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    String methodName = method.getName();
    if (methodName.equals("getAccounts")) {
      bank.setAccounts(getAccountsFromDatabase());
```

[9] Patrz [CGLIB], [ASM] i [Javassist].

```
        return bank.getAccounts();
      } else if (methodName.equals("setAccounts")) {
        bank.setAccounts((Collection<Account>) args[0]);
        setAccountsToDatabase(bank.getAccounts());
        return null;
      } else {
        ...
      }
    }

    // Tutaj sporo szczegółów:
    protected Collection<Account> getAccountsFromDatabase() { ... }
    protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}

// W innym miejscu...

Bank bank = (Bank) Proxy.newProxyInstance(
  Bank.class.getClassLoader(),
  new Class[] { Bank.class },
  new BankProxyHandler(new BankImpl()));
```

Zdefiniowaliśmy interfejs `Bank`, który został *osłonięty* za pomocą pośrednika, oraz *Plain-Old Java Object* (POJO), `BankImpl`, implementujący logikę biznesową (obiekty POJO zostaną przedstawione w dalszej części rozdziału).

API pośrednika wymaga zastosowania obiektu `InvocationHandler`, który przekazuje wywołania metod do implementacji `Bank`. Nasz `BankProxyHandler` korzysta z API refleksji do odwzorowania ogólnych wywołań metod na metody w `BankImpl`.

Mamy tu bardzo dużo kodu i jest on dosyć skomplikowany nawet dla prostego przypadku[10]. Użycie jednej z bibliotek manipulujących kodem wynikowym jest podobnie złożone. Taka objętość kodu i jego skomplikowanie to dwie poważne wady pośredników. Powodują one, że trudniej tworzyć czysty kod! Dodatkowo pośredniki nie zapewniają mechanizmów specyfikowania dostępnych systemowo „punktów" obsługi, które są niezbędne w prawdziwym rozwiązaniu AOP[11].

Czyste biblioteki Java AOP

Na szczęście większość zadań pośredników może być obsłużona automatycznie przez narzędzia. Pośredniki są używane wewnętrznie w kilku bibliotekach Java, na przykład Spring AOP oraz JBoss AOP, do implementowania aspektów w czystym języku Java[12]. W Spring logikę pisze się w postaci obiektów *Plain-Old Java Objects*. Obiekty POJO są ukierunkowane na swoją domenę. Nie mają one zależności z bibliotekami typu enterprise (ani innymi domenami). Dzięki temu są koncepcyjnie prostsze i łatwiejsze do testowania. Ich względna prostota ułatwia sprawdzenie, czy prawidłowo implementują odpowiednie problemy użytkownika, oraz ich utrzymanie i ewolucję kodu do obsługi kolejnych problemów w przyszłości.

---

[10] Bardziej szczegółowe przykłady API pośredników i przykłady jego zastosowania można znaleźć w [Goetz].

[11] AOP jest czasami mylone z technikami jego implementacji, takimi jak przechwytywanie metod i „osłanianie" za pomocą pośredników. Prawdziwą wartością AOP jest możliwość specyfikowania zachowań systemowych w spójny i modularny sposób.

[12] Patrz [Spring] oraz [JBoss]. „Czysta Java" oznacza, że nie zastosowano AspectJ.

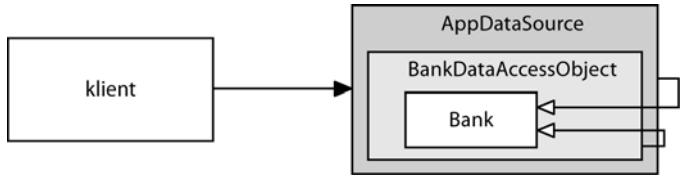
Wymaganą infrastrukturę aplikacji, w tym zagadnienia rozcięcia problemów, takie jak trwałość, transakcje, bezpieczeństwo, buforowanie, przełączanie w czasie awarii i tak dalej, dołączamy przez zastosowanie deklaratywnych plików konfiguracyjnych lub API. W wielu przypadkach faktycznie specyfikujemy aspekty bibliotek Spring lub JBoss, natomiast biblioteki obsługują mechanikę zastosowania pośredników Java lub bibliotek kodu wynikowego w sposób niewidoczny dla użytkownika. Deklaracje te sterują kontenerem wstrzykiwania zależności (DI), który tworzy na żądanie większość obiektów i łączy je ze sobą.

Na listingu 11.4 przedstawiony jest typowy fragment pliku konfiguracyjnego Spring V2.5, *app.xml*[13].

LISTING 11.4. Plik konfiguracyjny Spring 2.X

```
<beans>
  ...
  <bean id="appDataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close"
  p:driverClassName="com.mysql.jdbc.Driver"
  p:url="jdbc:mysql://localhost:3306/mydb"
  p:username="me"/>

  <bean id="bankDataAccessObject"
  class="com.example.banking.persistence.BankDataAccessObject"
  p:dataSource-ref="appDataSource"/>

  <bean id="bank"
  class="com.example.banking.model.Bank"
  p:dataAccessObject-ref="bankDataAccessObject"/>
  ...
</beans>
```

Każdy „bean" jest podobny do jednej części lalki matrioszki — obiekt domeny `Bank` posiada jako pośrednika (czyli jest opakowany przez) obiekt dostępu do danych (DAO), który z kolei ma jako pośrednika sterownik JDBC źródła danych (patrz rysunek 11.3).



RYSUNEK 11.3. „Lalka matrioszka" z dekoratorów

Klient uważa, że wykonuje metodę `getAccounts()` obiektu `Bank`, ale w rzeczywistości komunikuje się z najbardziej zewnętrznym ze zbioru *dekoratorów*[14] rozszerzających podstawowe funkcje POJO `Bank`. Możemy również dodać kolejne dekoratory dla transakcji, buforowania i innych mechanizmów.

---

[13] Zaadaptowany z *http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25*.

[14] [GOF].

W aplikacji konieczne jest użycie tylko kilku wierszy kodu do odpytania kontenera DI o obiekty najwyższego poziomu w systemie, zgodnie z definicją w pliku XML.

```
XmlBeanFactory bf =
  new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");
```

Dzięki temu, że wymaganych jest tak niewiele wierszy kodu specyficznego dla Spring, *aplikacja jest niemal całkowicie odłączona od Spring*, co eliminuje problem ścisłego złączenia z systemami takimi jak EJB2.

Choć XML może być rozwlekły i mało czytelny[15], „zasady" zdefiniowane w tych plikach konfiguracyjnych są prostsze niż skomplikowana logika pośredników i aspektów, która jest ukrywana i tworzona automatycznie. Tego typu architektura jest tak kusząca, że biblioteki podobne do Spring doprowadziły do całkowitego przebudowania standardu EJB w wersji 3. EJB w większości korzysta z modelu Spring deklaratywnego wspierania rozcięcia problemów, z użyciem plików konfiguracyjnych XML oraz (lub) adnotacji Java 5.

Na listingu 11.5 przedstawiony jest nasz obiekt `Bank` zapisany w EJB3[16].

LISTING 11.5. EJB Bank w EJB3

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private int id;

  @Embeddable // Obiekt "wbudowany" w wiersz bazy danych dla obiektu Bank.
  public class Address {
    protected String streetAddr1;
    protected String streetAddr2;
    protected String city;
    protected String state;
    protected String zipCode;
  }

  @Embedded
  private Address address;

  @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
             mappedBy="bank")
  private Collection<Account> accounts = new ArrayList<Account>();

  public int getId() {
    return id;
  }

  public void setId(int id) {
```

[15] Przykład może być uproszczony z użyciem mechanizmów wykorzystujących *konwencję przed konfiguracją* oraz adnotacje Java 5 do redukcji ilości wymaganej logiki „połączeń".

[16] Zaadaptowany z *http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html*.

```
    this.id = id;
  }

  public void addAccount(Account account) {
    account.setBank(this);
    accounts.add(account);
  }

  public Collection<Account> getAccounts() {
    return accounts;
  }

  public void setAccounts(Collection<Account> accounts) {
    this.accounts = accounts;
  }
}
```

Ten kod jest znacznie czystszy od wcześniejszego kodu EJB2. Niektóre szczegóły encji nadal się w nim znajdują, umieszczone w adnotacjach. Jednak dzięki temu, że żadna informacja nie znajduje się poza adnotacjami, kod jest czysty, jasny i przez to łatwy do testowania i utrzymania.

Jeżeli będzie to wymagane, niektóre lub wszystkie informacje dotyczące trwałości z adnotacji mogą być przeniesione do deskryptorów instalacji XML, co pozwoli pozostawić czyste obiekty POJO. Jeżeli szczegóły odwzorowania trwałości nie zmieniają się często, wiele zespołów wybiera korzystanie z adnotacji, ponieważ ten mechanizm ma znacznie mniej wad w porównaniu z inwazyjnym sposobem programowania w EJB2.

Aspekty w AspectJ

Na koniec przedstawione zostanie najbogatsze narzędzie rozdzielania problemów za pomocą języka AspectJ[17] — rozszerzenie języka Java, zapewniające „pierwszej klasy" obsługę aspektów do konstrukcji modularności. Rozwiązanie oparte na użyciu czystego języka Java, zapewniane przez Spring AOP oraz JBoss AOP, w zupełności wystarcza w znaczącej większości przypadków wykorzystania aspektów. Jednak AspectJ zapewnia bardzo przydatny i zaawansowany zestaw narzędzi do rozdzielania problemów. Wadą AspectJ jest potrzeba zastosowania kilku nowych narzędzi i nauczenia się nowych konstrukcji języka i idiomów.

Problemy z adaptacją zostały częściowo ograniczone przez ostatnio wprowadzoną „postać adnotacji" dla AspectJ, w której adnotacje Java 5 zostały użyte do definiowania aspektów z użyciem czystego kodu Java. Dodatkowo Spring Framework posiada kilka funkcji, które znacznie ułatwiają zastosowanie aspektów bazujących na adnotacjach w przypadku zespołów mających niewielkie doświadczenie w AspectJ.

Pełne omówienie rozszerzenia AspectJ wykracza poza zakres tej książki. Więcej informacji na ten temat można znaleźć w [AspectJ], [Colyer] oraz [Spring].

---

[17] Patrz [AspectJ] oraz [Colyer].

Testowanie architektury systemu

Siła rozdzielania problemów z użyciem metod aspektowych nie może być niedoceniana. Jeżeli możemy napisać logikę domeny naszej aplikacji z użyciem obiektów POJO, odłączonych na poziomie kodu od problemów architektury, to możliwe jest *testowanie* naszej architektury. Można również w razie potrzeby poddawać ją zmianom, z prostej tworząc całkiem skomplikowaną, przez adaptowanie nowych technologii. Nie jest konieczne wykonywanie wielkiego projektu od podstaw[18] (ang. *Big Design Up Front* — BDUF). W rzeczywistości projektowanie BDUF jest nawet szkodliwe, ponieważ uniemożliwia adaptowanie zmian m.in. z powodu psychologicznego oporu autora przed zaprzepaszczeniem wcześniejszej pracy oraz z powodu sposobu, w jaki wybory architektoniczne wpływają na dalsze myślenie o projekcie.

Architekci budynków muszą korzystać z BDUF, ponieważ nie da się wprowadzać radykalnych zmian architektonicznych w dużych strukturach fizycznych w czasie trwania ich budowy[19]. Jednak oprogramowanie posiada własną *fizykę*[20], w której możliwe jest wprowadzanie radykalnych zmian, *jeżeli* struktura oprogramowania efektywnie rozdziela problemy.

Oznacza to, że można rozpocząć projekt oprogramowania, korzystając z „naiwnie prostej", ale rozłączonej architektury, i dostarczając szybko użytkownikom działające rozwiązania, a następnie dodając coraz więcej elementów infrastruktury w procesie skalowania w górę. Niektóre z największych na świecie witryn WWW osiągnęły wysoką dostępność i wydajność przez zastosowanie skomplikowanych mechanizmów buforowania danych, bezpieczeństwa, wirtualizacji itp. w sposób efektywny i elastyczny, ponieważ minimalnie sprzężone projekty były odpowiednio *proste* na każdym poziomie abstrakcji.

Oczywiście nie oznacza to, że idziemy w kierunku projektu „bez steru". Mamy pewne oczekiwania co do zakresu, celów i harmonogramu projektu, jak również ogólną wizję struktury wynikowego systemu. Jednak musimy utrzymywać możliwość zmiany kierunku w odpowiedzi na zmieniające się okoliczności.

Wczesna architektura EJB jest jednym z najlepiej znanych API, które są przeprojektowane i nie umożliwiają rozdzielenia problemów. Nawet dobrze zaprojektowane API może być przesadne, jeżeli nie jest ono naprawdę potrzebne. Dobre API powinno w większości przypadków *znikać* z widoku, aby zespół spędzał większość swojego czasu na implementowaniu rozwiązań dla użytkownika. W przeciwnym razie ograniczenia architektury będą zmniejszały efektywność dostarczania klientom optymalnych rozwiązań.

Podsumujmy ten długi opis:

> *Optymalna architektura systemu składa się z modularnych domen problemów; każda z nich jest implementowana z użyciem zwykłych obiektów, np. POJO. Osobne domeny są integrowane ze sobą przy użyciu minimalnie inwazyjnych narzędzi aspektowych lub podobnych. Architektura ta może być testowana, podobnie jak kod.*

---

[18] Nie należy mylić tego z dobrą praktyką projektowania od podstaw — BDUF jest praktyką projektowania *wszystkiego* od początku, przed rozpoczęciem implementowania czegokolwiek.

[19] Nadal stosowane jest iteracyjne analizowanie i dyskusja nad detalami, nawet po rozpoczęciu budowy.

[20] Termin *fizyka oprogramowania* został po raz pierwszy użyty w [Kolence].

Optymalizacja podejmowania decyzji

Modularność i rozdzielenie problemów umożliwiają decentralizowane zarządzanie i podejmowanie decyzji. W odpowiednio dużym systemie, niezależnie od tego, czy jest to miasto, czy projekt oprogramowania, jedna osoba nie może podejmować wszystkich decyzji.

Wszyscy wiemy, że najlepiej rozdzielać odpowiedzialność między najbardziej kwalifikowane osoby. Często zapominamy, że również najlepiej *opóźniać decyzję do ostatniego możliwego momentu*. Nie jest to lenistwo ani brak odpowiedzialności — pozwala to nam dokonać wyboru z użyciem najbardziej aktualnych informacji. Zbyt wczesna decyzja jest decyzją podjętą na podstawie nieoptymalnych informacji. Mamy w takim przypadku znacznie mniej informacji od klienta, wiedzy dotyczącej projektu i doświadczenia z naszymi wyborami implementacyjnymi.

> *Solidność zapewniana przez system POJO z modularnymi problemami pozwala na podejmowanie optymalnych decyzji na czas, przy czym decyzje te bazują na najbardziej aktualnej wiedzy. Zmniejszona jest również złożoność tych decyzji.*

Korzystaj ze standardów, gdy wnoszą realną wartość

Fantastycznie jest patrzeć na plac budowy, zarówno ze względu na szybkość, z jaką powstają nowe budynki (nawet w środku zimy), jak i na nadzwyczajne projekty, jakie są możliwe do realizacji z użyciem dzisiejszych technologii. Budownictwo jest dojrzałym przemysłem korzystającym z wysoce zoptymalizowanych części, metod i standardów, które ewoluowały przez setki lat.

Wiele zespołów używało architektury EJB2, ponieważ był to standard, nawet jeżeli wystarczyło zastosować prostsze projekty. Spotkałem się z zespołami, które miały obsesję na punkcie *mocno przereklamowanych* standardów i przestawały się skupiać na dostarczaniu wartości swoim klientom.

> *Standardy pozwalają na wielokrotne wykorzystywanie pomysłów i komponentów, zatrudnianie osób o odpowiednim doświadczeniu, hermetyzację dobrych pomysłów i łączenie ze sobą komponentów. Jednak proces tworzenia standardów może zajmować zbyt wiele czasu, przez co część z nich traci kontakt z faktycznymi potrzebami ich użytkowników — potrzebami, które te standardy miały zaspokajać.*

Systemy wymagają języków dziedzinowych

Budownictwo, podobnie jak większość dziedzin technologii, wytworzyło bogaty język ze słownictwem, idiomami oraz wzorcami[21], które pozwalają na przekazywanie istotnych informacji w jasny i zwięzły sposób. W przypadku oprogramowania ostatnio odnowiło się zainteresowanie tworzeniem *języków dziedzinowych* (DSL)[22] — osobnych, niewielkich języków skryptowych lub API w standardowych językach, pozwalających na napisanie kodu, który czyta się jak porządnie skomponowaną prozę.

---

[21] Na informatykę największy wpływ miała praca [Alexander].

[22] Patrz na przykład [DSL]. [JMock] jest dobrym przykładem API Java do tworzenia DSL.

Dobry język DSL minimalizuje „przepaść komunikacyjną" pomiędzy koncepcją domeny a implementującym ją kodem, podobnie jak solidne praktyki optymalizują komunikację wewnątrz zespołu oraz z udziałowcami projektu. Jeżeli implementujemy logikę domeny w tym samym języku, którego używa ekspert w zakresie domeny, zmniejszamy ryzyko niewłaściwej konwersji domeny na implementację.

Używane efektywnie języki DSL podnoszą poziom abstrakcji ponad idiomy kodu i wzorce projektowe. Pozwalają programistom na odcyfrowanie przeznaczenia kodu na odpowiednim poziomie abstrakcji.

> *Języki dziedzinowe pozwalają wszystkim poziomom abstrakcji i wszystkim domenom w aplikacji na korzystanie z POJO — począwszy od zasad wysokiego poziomu, na szczegółach niskiego poziomu skończywszy.*

Zakończenie

Systemy również muszą być czyste. Inwazyjna architektura przytłacza logikę domeny i niekorzystnie wpływa na jej solidność. Gdy logika domeny jest zaburzona, spada jakość działania systemu, ponieważ błędy znacznie łatwiej ukrywają się w aplikacji, a implementacja rozwiązań staje się trudniejsza. W efekcie spada wydajność i zalety TDD są tracone.

Intencje powinny być jasne na wszystkich poziomach abstrakcji. Zdarza się to tylko wtedy, gdy korzystamy z POJO oraz mechanizmów zbliżonych do aspektowych w celu nieinwazyjnego zastosowania innych implementacji problemów.

Niezależnie od tego, czy projektujemy systemy, czy indywidualne moduły, nigdy nie należy zapominać o *korzystaniu z najprostszych skutecznych mechanizmów*.

Bibliografia

[Alexander]: Christopher Alexander, *A Timeless Way of Building*, Oxford University Press, New York 1979.

[AOSD]: Aspect-Oriented Software Development, *http://aosd.net*.

[ASM]: Strona główna ASM, *http://asm.objectweb.org*.

[AspectJ]: *http://eclipse.org/aspectj*.

[CGLIB]: Code Generation Library, *http://cglib.sourceforge.net*.

[Colyer]: Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, *Eclipse AspectJ*, Person Education, Inc., Upper Saddle River, New York 2005.

[DSL]: Język dziedzinowy, *http://en.wikipedia.org/wiki/Domainspecific_programming_language* lub *http://pl.wikipedia.org/wiki/J%C4%99zyk_dziedzinowy*.

[Fowler]: *Inversion of Control Containers and the Dependency Injection pattern*, *http://martinfowler.com/articles/injection.html*.

[Goetz]: Brian Goetz, *Java Theory and Practice: Decorating with Dynamic Proxies*, *http://www.ibm.com/developerworks/java/library/j-jtp08305.html*.

[Javassist]: Strona główna Javassist, *http://www.csg.is.titech.ac.jp/~chiba/javassist*.

[JBoss]: Strona główna JBoss, *http://jboss.org*.

[JMock]: Jmock — A Lightweight Mock Object Library for Java, *http://jmock.org*.

[Kolence]: Kenneth W. Kolence, *Software physics and computer performance measurements*, *Proceedings of the ACM annual conference — Volume 2*, Boston, Massachusetts, s. 1024 – 1040, 1972.

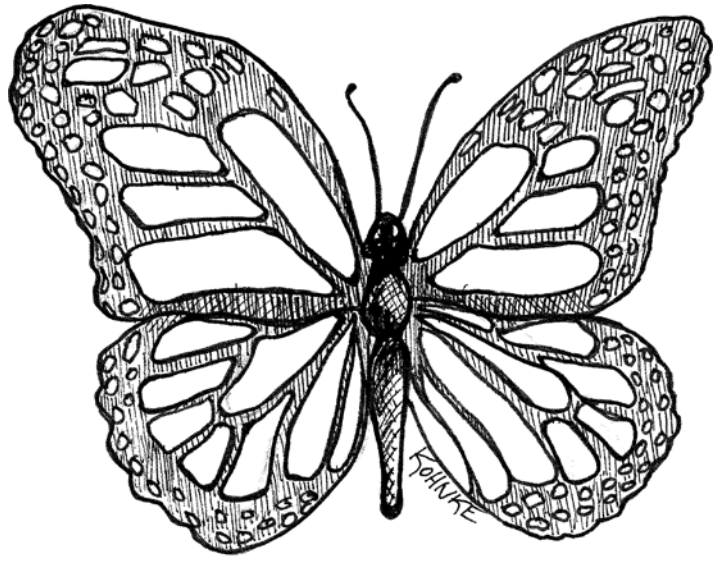
[Spring]: The Spring Framework, *http://www.springframework.org*.

[Mezzaros07]: Gerard Mezzaros, *XUnit Patterns*, Addison-Wesley 2007.

[GOF]: Gamma i inni, *Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

Powstawanie projektu

Jeff Langr



Uzyskiwanie czystości projektu przez jego rozwijanie

Co by było, gdyby dostępne były cztery proste zasady, które pomagałyby tworzyć dobre projekty? Co by było, gdyby korzystanie z tych zasad dawało wgląd w strukturę i projekt kodu, ułatwiając stosowanie takich zasad, jak SRP i DIP? Co by było, gdyby te cztery zasady ułatwiały *powstawanie* dobrych projektów?

Wielu z nas uważa, że cztery zasady *prostego projektu*[1] opracowane przez Kenta Becka są znaczącą pomocą przy tworzeniu dobrze zaprojektowanego oprogramowania.

[1] [XPE].

Według Kenta projekt jest „prosty", jeżeli spełnia następujące zasady:

- Przechodzi wszystkie testy.
- Nie zawiera powtórzeń.
- Wyraża intencje programisty.
- Minimalizuje liczbę klas i metod.

Zasady te są zamieszczone zgodnie z hierarchią ważności.

Zasada numer 1 prostego projektu — system przechodzi wszystkie testy

Po pierwsze i najważniejsze, projekt musi dać w wyniku system działający w zamierzony sposób. System może mieć doskonały projekt na papierze, ale jeżeli nie istnieje prosty sposób na weryfikowanie, czy działa on w oczekiwany sposób, to cały projekt papierowy jest niewiele warty.

System, który jest dokładnie testowany i przechodzi za każdym razem wszystkie testy, jest systemem testowalnym. Można przytoczyć oczywiste, ale ważne zdanie: systemy, które nie są testowalne, nie są również weryfikowalne. Z kolei system, którego nie można zweryfikować, nie powinien być nigdy instalowany.

Na szczęście zadanie doprowadzenia do testowalności systemu kieruje nas w stronę projektu, w którym klasy są małe i jednozadaniowe. Po prostu łatwiej jest testować klasy, które spełniają SRP. Im więcej testów napiszemy, tym częściej będziemy tworzyć konstrukcje łatwe w testowaniu. Tak więc zapewnienie pełnej testowalności systemu pomaga w tworzeniu lepszych projektów.

Ścisłe sprzężenie utrudnia pisanie testów. Podobnie, im więcej testów napiszemy, tym częściej będziemy korzystać z zasad takich jak DIP i narzędzi takich jak interfejsy wstrzykiwania zależności i abstrakcji zapewniających minimalizację sprzężeń. Nasze projekty staną się jeszcze lepsze.

Co ciekawe, działanie zgodnie z prostą i oczywistą zasadą mówiącą, że musimy mieć testy i stale je wykonywać, wpływa na zgodność systemu z podstawowymi celami obiektowości — niskiego sprzężenia i wysokiej spójności. Pisanie testów prowadzi do tworzenia lepszych projektów.

Zasady numer 2 – 4 prostego projektu — przebudowa

Gdy mamy zbudowane testy, jesteśmy przygotowani do zapewnienia czystości kodu i klas. Realizujemy to przez przyrostową przebudowę kodu. Po dodaniu każdego wiersza kodu powinniśmy się zatrzymać i pomyśleć o nowym projekcie. Czy właśnie go nie zepsuliśmy? Jeżeli tak, należy poprawić kod i wykonać testy w celu pokazania, że niczego nie uszkodziliśmy. *W rzeczywistości posiadanie tych testów eliminuje obawę, że proces czyszczenia kodu uszkodzi go!*

W czasie operacji przebudowy możemy stosować dowolne techniki z bogatej wiedzy na temat dobrych projektów oprogramowania. Możemy zwiększać spójność projektu, zmniejszać sprzężenia, rozdzielać problemy, modularyzować mechanizmy, skracać funkcje i klasy, tworzyć lepsze nazwy i tak dalej. Tu również stosujemy trzy ostatnie zasady prostego projektu — eliminujemy powtórzenia, zapewniamy wyrazistość kodu i minimalizujemy liczbę klas i metod.

Brak powtórzeń

Powtórzenia są wrogiem publicznym w dobrze zaprojektowanym systemie. Reprezentują one dodatkową pracę, dodatkowe ryzyko i dodatkową niepotrzebną złożoność. Powtórzenia pojawiają się w wielu formach. Wiersze kodu wyglądające dokładnie tak samo są oczywiście powtórzeniami. Wiersze kodu, które są podobne, często mogą być przekształcone tak, aby wyglądały bardziej podobnie, dzięki czemu mogą być one łatwiej przebudowane. Powtórzenia mogą również istnieć w innych formach, jak na przykład powtórzenia implementacji. Możemy mieć dwie metody w kolekcji klas:

```
int size() {}
boolean isEmpty() {}
```

Możemy mieć osobne implementacje dla każdej z tych metod. Metoda `isEmpty` może zwracać wartość logiczną, natomiast `size` wartość licznika. Można również wyeliminować powtórzenia przez związanie `isEmpty` z definicją `size`:

```
boolean isEmpty() {
  return 0 == size();
}
```

Tworzenie czystego systemu wymaga eliminowania powtórzeń, nawet jeżeli jest to tylko kilka wierszy kodu. Weźmy pod uwagę następujący kod:

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
    return;
  float scalingFactor = desiredDimension / imageDimension;
  scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
  RenderedOp newImage = ImageUtilities.getScaledImage(
    image, scalingFactor, scalingFactor);
  image.dispose();
  System.gc();
  image = newImage;
}
public synchronized void rotate(int degrees) {
  RenderedOp newImage = ImageUtilities.getRotatedImage(
    image, degrees);
  image.dispose();
  System.gc();
  image = newImage;
}
```

Aby zachować czystość systemu, powinniśmy wyeliminować małe powtórzenia pomiędzy metodami `scaleToDimension` a `rotate`:

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
    return;
  float scalingFactor = desiredDimension / imageDimension;
  scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);
  replaceImage(ImageUtilities.getScaledImage(
    image, scalingFactor, scalingFactor));
}

public synchronized void rotate(int degrees) {
  replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}

private void replaceImage(RenderedOp newImage) {
  image.dispose();
  System.gc();
  image = newImage;
}
```

Gdy wyodrębniamy wspólne części na tak niskim poziomie, zaczynamy rozpoznawać naruszenia SRP. Możemy więc przenieść nowe, wyekstrahowane metody do innej klasy. Pozwala to poprawić przejrzystość kodu. Inna osoba z zespołu może rozpoznać możliwość utworzenia dalszej abstrakcji w nowej metodzie i ponownie ją wykorzystać w innym kontekście. Takie „ponowne użycie w małej skali" może spowodować znaczące zmniejszenie złożoności systemu. Jeżeli będziemy widzieć, w jaki sposób ponownie wykorzystywać kod w małej skali, to znacznie prostsze będzie jego wykorzystanie w większej skali.

Zastosowanie wzorca szablonu metody[2] (ang. *Template Method*) jest często stosowaną techniką usuwania powtórzeń na wysokim poziomie. Na przykład:

```
public class VacationPolicy {
  public void accrueUSDivisionVacation() {
  // Kod obliczający urlopy, bazujący na godzinach przepracowanych do daty.
  // ...
  // Kod sprawdzający, czy urlop spełnia minimalne wymagania z USA.
  // ...
  // Kod zapisujący urlop w systemie płacowym.
  // ...
   }

   public void accrueEUDivisionVacation() {
  // Kod obliczający urlopy, bazujący na godzinach przepracowanych do daty.
  // ...
  // Kod sprawdzający, czy urlop spełnia minimalne wymagania przepisów Unii Europejskiej.
  // ...
  // Kod zapisujący urlop w systemie płacowym.
  // ...
   }
}
```

Kod w metodach `accrueUSDivisionVacation` oraz `accrueEuropeanDivisionVacation` jest w większości taki sam, z wyjątkiem obliczania wartości minimalnych. Ten fragment algorytmu zmienia się w zależności od typu pracownika.

---

[2] [GOF].

Te oczywiste powtórzenia można wyeliminować przez zastosowanie wzorca szablonu metody.

```
abstract public class VacationPolicy {
  public void accrueVacation() {
    calculateBaseVacationHours();
    alterForLegalMinimums();
    applyToPayroll();
  }
  private void calculateBaseVacationHours() { /* ... */ };
  abstract protected void alterForLegalMinimums();
  private void applyToPayroll() { /* ... */ };
}
public class USVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // Kod specyficzny dla USA.
  }
}
public class EUVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
  // Kod specyficzny dla UE.
  }
}
```

Dziedziczenie wypełnia „dziurę" w algorytmie `accrueVacation`, dostarczając jedyny fragment informacji, który nie jest powtarzany.

Wyrazistość kodu

Większość z nas ma doświadczenie w pracy z zawiłym kodem. Większość z nas napisała nieco takiego kodu. Łatwo napisać kod, który sami możemy zrozumieć, ponieważ w czasie pisania dokładnie rozumiemy problem, który próbujemy rozwiązać. Inne osoby, które konserwują kod, nie muszą mieć tak głębokiego rozeznania.

Większość kosztu oprogramowania przypada na jego długoterminowe utrzymanie. Aby zminimalizować potencjalne defekty, jakie są wprowadzane przez zmiany, niezbędne jest zrozumienie tego, co realizuje system. Wraz ze wzrostem stopnia skomplikowania systemu coraz więcej czasu zajmuje programistom zrozumienie jego działania, a dodatkowo istnieje stale zwiększające się niebezpieczeństwo niewłaściwego zrozumienia. Dlatego kod powinien jasno wyrażać intencje autora. Im jaśniejszy jest kod napisany przez autora, tym mniej czasu inne osoby będą potrzebowały na jego zrozumienie. Pozwala to na ograniczenie defektów i zmniejszenie kosztu utrzymania kodu.

Co zatem można zrobić, by kod był czytelny?

Można wybierać sugestywne nazwy. Chcemy móc usłyszeć nazwę klasy lub funkcji i nie być zaskoczonym, gdy odkryjemy ich odpowiedzialności.

Można również tworzyć niewielkie funkcje i klasy. Małe klasy i funkcje są zwykle łatwiejsze do nazwania, napisania i zrozumienia.

Można również korzystać ze standardowej nomenklatury. Wzorce projektowe dla przykładu są w większości przeznaczone do poprawiania komunikacji i wyrazistości kodu. Przez stosowanie standardowych nazw wzorców, takich jak `Command` lub `Visitor` w nazwach klas implementujących te wzorce, możemy jasno opisywać nasz projekt dla innych programistów.

Dobrze napisane testy jednostkowe są również ekspresyjne. Podstawowym celem testów jest zapewnienie dokumentacji przez przykład. Ten, kto czyta nasze testy, powinien móc szybko zrozumieć przeznaczenie klasy.

Jednak najważniejszym sposobem na osiągnięcie wyrazistości kodu jest *próbowanie*. Zbyt często zajmujemy się wyłącznie tym, aby kod działał prawidłowo, i przechodzimy do następnego problemu, nie zaprzątając sobie głowy kwestią czytelności kodu. Należy pamiętać, że najprawdopodobniej tą następną osobą będziemy my sami.

Dlatego warto być dumnym ze swojej pracy. Warto spędzić nieco czasu nad każdą funkcją i klasą — wybrać lepsze nazwy, podzielić dużą funkcję na mniejsze i ogólnie przywiązywać większą wagę do tego, co tworzymy. Uwaga jest cennym zasobem.

Minimalne klasy i metody

Nawet tak podstawowe koncepcje, jak eliminacja powtórzeń, wyrazistość kodu czy też SRP, mogą być pojmowane zbyt dosłownie. W procesie tworzenia małych klas i metod możemy utworzyć zbyt wiele niewielkich klas i metod. Dlatego zasada ta sugeruje, aby równocześnie zachowywać niewielką liczbę funkcji i klas.

Duża liczba klas i metod jest czasami wynikiem bezcelowego dogmatyzmu. Weźmy jako przykład standard kodowania, który zakłada tworzenie interfejsu dla każdej klasy, lub programistę, który uważa, że pola i operacje muszą być zawsze rozdzielone na klasy danych i klasy operacji. Nie tędy droga.

Naszym celem jest zachowanie możliwie małego systemu, przy jednoczesnym ograniczaniu liczby funkcji i klas. Należy jednak pamiętać, że zasada ta ma najniższy priorytet z czterech zasad prostego projektu. Tak więc choć utrzymanie niskiej liczby klas i funkcji jest ważne, ważniejsze jest posiadanie testów, eliminowanie powtórzeń i wyrazistość kodu.

Zakończenie

Czy istnieje zbiór prostych zasad, które mogą zastąpić doświadczenie? Oczywiście nie. Z drugiej strony, praktyki opisane w tym rozdziale i całej książce są wykrystalizowaną postacią wielu dekad doświadczenia zdobywanego przez autorów. Korzystanie z praktyki prostego projektu może i powinno zachęcać i skłaniać programistów do stosowania dobrych praktyk oraz wzorców, których w innym przypadku uczyliby się latami.

Bibliografia

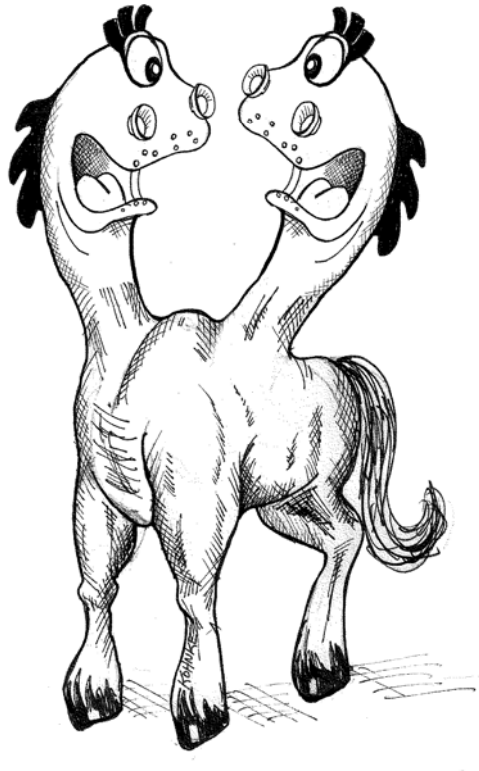
[XPE]: Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley 1999.

[GOF]: Gamma i inni, *Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

ROZDZIAŁ 13.

Współbieżność

Brett L. Schuchert



Obiekty są abstrakcją przetwarzania.
Wątki są abstrakcją harmonogramów.

James O. Coplien[1]

[1] Z korespondencji prywatnej.

Pisanie czystych programów współbieżnych jest trudne — bardzo trudne. Znacznie łatwiej pisać kod, który wykonuje się w jednym wątku. Równie łatwo można pisać kod wielowątkowy, który wygląda świetnie na pierwszy rzut oka, ale zawiera błędy na niższych poziomach. Taki kod działa prawidłowo do momentu, gdy system zostanie poddany większemu obciążeniu.

W tym rozdziale przedstawimy potrzebę programowania współbieżnego oraz związane z tym trudności. Przedstawimy kilka zaleceń pozwalających poradzić sobie z tymi trudnościami oraz pisać czysty kod współbieżny. Na koniec zajmiemy się problemami związanymi z testowaniem kodu współbieżnego.

Współbieżność kodu jest złożonym zagadnieniem, które może być tematem osobnej książki. W tej książce przedstawimy go w dwóch etapach: w niniejszym rozdziale zawarte jest wprowadzenie do zagadnienia, a bardziej szczegółowy samouczek zamieściliśmy w dodatku A, „Współbieżność II". Jeżeli Czytelnik jest po prostu ciekaw zagadnień związanych ze współbieżnością, to ten rozdział jest wystarczający. Jeżeli konieczne jest dogłębne zrozumienie współbieżności, lektura samouczka jest niezbędna.

W jakim celu stosować współbieżność?

Współbieżność jest strategią rozdzielania. Pomaga nam oddzielić to, *co* jest wykonywane, od tego, *kiedy* jest wykonywane. W aplikacji jednowątkowej *co* i *kiedy* są tak ściśle powiązane, że stan całej aplikacji może być często określony wyłącznie na podstawie zapisu stosu. Programista debugujący taki system może ustawić punkty zatrzymania lub ich sekwencję i będzie *wiedział*, jaki jest stan systemu w czasie, gdy zostanie zatrzymany.

Rozłączenie tego, *co* jest wykonywane, od tego, *kiedy* jest wykonywane, może znakomicie poprawić zarówno wydajność, jak i strukturę aplikacji. Ze strukturalnego punktu widzenia aplikacja jest podobna do wielu współpracujących ze sobą komputerów zamiast jednej wielkiej pętli głównej. Pozwala to łatwiej zrozumieć system i oferuje wiele wydajnych sposobów na rozdzielenie problemów.

Jako przykład weźmy standardowy model aplikacji WWW z użyciem serwletów. Systemy te działają pod parasolem kontenera WWW lub EJB, który częściowo *zarządza* za nas współbieżnością. Serwlety są wykonywane asynchronicznie w momencie przyjęcia żądania WWW. Programista serwletu nie musi zarządzać wszystkimi przychodzącymi żądaniami. *W zasadzie* każdy serwlet wykonuje się w osobnym małym świecie i jest odłączony od działania innych serwletów.

Oczywiście, jeżeli byłoby to proste, rozdział ten byłby niepotrzebny. W rzeczywistości oddzielenie zapewniane przez kontenery WWW jest dalekie od doskonałości. Programiści serwletów muszą być uważni i ostrożni przy programowaniu, aby ich współbieżne programy były prawidłowe. Pomimo tego strukturalne zalety modelu serwletów są znaczne.

Jednak struktura nie jest jedynym motywem wykorzystywania współbieżności. Niektóre systemy mają narzucone ograniczenia na czas odpowiedzi i przepustowość, które wymagają ręcznego kodowania rozwiązań współbieżnych. Dla przykładu weźmy jednowątkowy agregator informacji, który zbiera informacje z wielu różnych witryn WWW i łączy te informacje w dzienne podsumowanie.

Ponieważ system jest jednowątkowy, odpytuje kolejne witryny, kończąc przeszukiwanie jednej przed rozpoczęciem następnej. Jedno uruchomienie musi się zakończyć w czasie krótszym niż 24 godziny. Jednak gdy dodawanych jest coraz więcej witryn WWW, czas ten rośnie, i w końcu zebranie wszystkich danych zajmuje ponad 24 godziny. Aplikacja jednowątkowa potrzebuje bardzo dużo czasu na oczekiwanie zakończenia operacji wejścia-wyjścia przez gniazdo WWW. Możemy poprawić wydajność takiej aplikacji przez użycie algorytmu wielowątkowego, który jednocześnie przegląda więcej niż jedną witrynę WWW.

Innym przykładem zastosowania współbieżności może być system obsługujący jednego użytkownika jednocześnie, wymagający tylko sekundy na obsłużenie użytkownika. Taki system dosyć szybko reaguje w przypadku wielu użytkowników, ale wraz ze wzrostem ich liczby czas odpowiedzi systemu zwiększa się. Żaden użytkownik nie ma ochoty czekać w kolejce za 150 innymi! Możemy poprawić czas odpowiedzi takiego systemu przez obsługę wielu użytkowników jednocześnie.

Współbieżność może również znaleźć zastosowanie w systemie interpretującym duże zbiory danych, który może dać pełne rozwiązanie po przetworzeniu wszystkich zbiorów. Prawdopodobnie każdy zbiór danych jest przetwarzany przez inny komputer, więc wiele zbiorów danych jest przetwarzanych równolegle.

Mity i nieporozumienia

Mamy więc kuszące powody stosowania współbieżności. Jednak jak wcześniej wspomnieliśmy, współbieżność jest *trudna*. Jeżeli nie będziemy ostrożni, możemy doprowadzić do wielu nieprzyjemnych sytuacji. Przeanalizujmy poniższe często spotykane mity i nieporozumienia:

- *Współbieżność zawsze poprawia wydajność.*

  Współbieżność może *czasami* poprawić wydajność aplikacji, ale tylko w przypadkach, gdy program traci dużo czasu na oczekiwaniu na wykonanie pewnych procesów i gdy ten czas można podzielić pomiędzy wiele wątków lub wiele procesorów.

- *Projekt nie zmienia się, jeżeli piszemy programy współbieżne.*

  W rzeczywistości projekt algorytmu współbieżnego może być całkowicie inny niż projekt systemu jednowątkowego. Rozdzielenie *co* od *kiedy* zwykle ma ogromny wpływ na strukturę systemu.

- *Zrozumienie problemów ze współbieżnością nie jest ważne w przypadku korzystania z takiego kontenera jak WWW lub EJB.*

  W rzeczywistości powinniśmy doskonale wiedzieć, co robi nasz kontener i jak chronić się przed problemami współbieżnej modyfikacji oraz zakleszczeń działania kodu, które zostaną przedstawione w dalszej części rozdziału.

Dalej mamy kilka bardziej wyważonych stwierdzeń związanych z pisaniem programów współbieżnych:

- *Współbieżność wymaga pewnych narzutów*, zarówno na wydajność, jak i na pisanie dodatkowego kodu.

- *Prawidłowa obsługa współbieżności jest złożona*, nawet w przypadku prostych problemów.

- *Błędy współbieżności często nie są powtarzalne*, więc zwykle są ignorowane jako jednorazowe[2], choć w rzeczywistości są prawdziwymi defektami.
- *Współbieżność często wymaga podstawowych zmian w strategii projektu.*

Wyzwania

Co powoduje, że programowanie współbieżne jest tak trudne? Weźmy jako przykład taką prostą klasę:

```
public class X {
  private int lastIdUsed;

  public int getNextId() {
    return ++lastIdUsed;
  }
}
```

Załóżmy, że tworzymy obiekt `X`, ustawiamy pole `lastIdUsed` na 42, a następnie współdzielimy ten obiekt między dwa wątki. Załóżmy teraz, że oba te wątki wywołują metodę `getNextId()`; istnieją trzy możliwe wyniki:

- wątek I otrzymuje wartość 43, wątek II otrzymuje wartość 44, `lastUsedId` jest równe 44;
- wątek I otrzymuje wartość 44, wątek II otrzymuje wartość 43, `lastUsedId` jest równe 44;
- wątek I otrzymuje wartość 43, wątek II otrzymuje wartość 43, `lastUsedId` jest równe 43.

Zaskakujący trzeci wynik[3] występuje, gdy dwa wątki napotkają się. Dzieje się to, ponieważ istnieje wiele możliwych ścieżek, które mogą być wybrane przez te dwa wątki w czasie wykonywania jednego wiersza kodu Java, a część z tych ścieżek generuje niewłaściwe wyniki. Ile jest takich różnych ścieżek? Aby rzetelnie odpowiedzieć na to pytanie, musimy zrozumieć, co robi kompilator JIT z wygenerowanym kodem i co w modelu pamięci Java jest uznawane za atomowe.

Dla tych dwóch wątków wykonujących metodę `getNextId` na podstawie właśnie wygenerowanego kodu bajtowego występuje 12 870 różnych możliwych ścieżek wykonania[4]. Jeżeli zmienimy typ zmiennej `lastIdUsed` z `int` na `long`, liczba możliwych ścieżek zwiększy się do 2 704 156. Oczywiście, większość ścieżek wygeneruje prawidłowe wyniki. Problemem jest jednak, że *niektóre nie dadzą prawidłowych wyników*.

Zasady obrony współbieżności

Przedstawimy teraz serię zasad i technik obrony naszych systemów przed problemami ze współbieżnym kodem.

---

[2] Promienie kosmiczne, szumy i tak dalej.

[3] Patrz „Kopiemy głębiej” w dodatku A.

[4] Patrz „Możliwe ścieżki wykonania” w dodatku A.

Zasada pojedynczej odpowiedzialności

Zasada SRP[5] mówi, że dana metoda (klasa, komponent) powinna mieć jeden powód do zmiany. Projekt współbieżności jest na tyle złożony, aby być powodem do zmiany, i dlatego warto go oddzielić od reszty kodu. Niestety, zbyt często implementacja szczegółów współbieżności jest wbudowywana bezpośrednio w pozostały kod produkcyjny. Warto wziąć pod uwagę następujące zagadnienia:

- *Kod związany ze współbieżnością ma własny cykl produkcji*, zmian i dostrajania.
- *Kod związany ze współbieżnością ma własne wyzwania*, które różnią się od tych z kodu niewspółbieżnego i często są trudniejsze.
- Liczba sposobów, na które źle napisany kod współbieżny może zawieść, powoduje, że jest on wystarczającym wyzwaniem, nawet bez otaczającego go kodu aplikacji.

Zalecenie: *Kod związany ze współbieżnością powinniśmy oddzielić od pozostałego kodu*[6].

Wniosek — ograniczenie zakresu danych

Jak widzieliśmy, dwa wątki modyfikujące to samo pole współużytkowanego obiektu mogą wpływać na siebie, powodując nieoczekiwane działanie. Jednym z rozwiązań jest zastosowanie słowa kluczowego `synchronized` w celu ochrony *sekcji krytycznej* kodu wykorzystującego współużytkowane obiekty. Ważne jest, aby ograniczać liczbę takich sekcji krytycznych. Im więcej mamy miejsc, w których modyfikowane są wspólne dane, tym większe prawdopodobieństwo, że:

- zapomnimy ochronić jedno lub więcej takich miejsc — w efekcie powodując uszkodzenie całego kodu modyfikującego te wspólne dane;
- wystąpi powtórzenie zadań wymaganych do upewnienia się, że wszystko jest efektywnie chronione (naruszenie zasady DRY[7]);
- trudno będzie określić źródło awarii, które i tak jest wystarczająco trudne do znalezienia.

Zalecenie: *Należy wziąć sobie do serca zagadnienie hermetyzacji danych i w znacznym stopniu ograniczyć dostęp do danych, które mogą być współużytkowane.*

Wniosek — korzystanie z kopii danych

Dobrym sposobem na eliminowanie współużytkowanych danych jest... unikanie ich współużytkowania. W niektórych przypadkach możliwe jest kopiowanie obiektów i traktowanie ich jako tylko do odczytu. W innych przypadkach może być możliwe kopiowanie obiektów, zbieranie wyników z wielu wątków w tych kopiach, a następnie łączenie wyników w jednym wątku.

---

[5] [PPP].

[6] Patrz „Przykład klient-serwer" w dodatku A.

[7] [PRAG].

Jeżeli istnieje prosty sposób na uniknięcie współużytkowania obiektów, wynikowy kod będzie ze znacznie mniejszym prawdopodobieństwem powodował błędy. Można się jednak obawiać o koszt tworzenia dodatkowych obiektów. Warto poeksperymentować w celu określenia, czy faktycznie jest to problemem. Jednak jeżeli wykorzystanie kopii obiektów pozwala na uniknięcie synchronizacji, oszczędności na uniknięciu koniecznej blokady zwykle będą większe niż narzut wymagany na dodatkowe tworzenie i usuwanie obiektu.

Wniosek — wątki powinny być na tyle niezależne, na ile to tylko możliwe

Warto rozważyć pisanie kodu wątków w taki sposób, aby każdy wątek działał we własnym świecie, nie współużytkując danych z pozostałymi wątkami. Każdy wątek przetwarza jedno żądanie klienta, a wszystkie wymagane dane pochodzą z niewspółużytkowanego źródła i są przechowywane jako zmienne lokalne. Powoduje to, że wątki te działają tak, jakby były jedynym wątkiem na świecie, i dlatego nie potrzebują synchronizacji.

Na przykład klasy dziedziczące po `HttpSevlet` otrzymują wszystkie dane jako parametry przekazane do metod `doGet` oraz `doPost`. Powoduje to, że każdy serwlet działa jako osobna maszyna. Dopóki kod w serwlecie korzysta wyłącznie ze zmiennych lokalnych, nie ma możliwości, aby wystąpiły błędy synchronizacji. Oczywiście, większość aplikacji korzystających z serwletów i tak korzysta ze współużytkowanych zasobów, na przykład połączeń z bazą danych.

Zalecenie: *Warto spróbować podzielić dane na niezależne podzbiory, na których mogą działać niezależne wątki, być może działające na różnych procesorach.*

Poznaj używaną bibliotekę

W stosunku do poprzedniej wersji język Java 5 oferuje wiele usprawnień programowania współbieżnego. Pisząc kod korzystający z wątków za pomocą języka Java 5, należy rozważyć następujące zagadnienia:

- Zastosowanie dostarczanych kolekcji bezpiecznych dla wątków.
- Użycie wykonawców do uruchamiania niezwiązanych ze sobą zadań.
- Stosowanie rozwiązań nieblokujących wszędzie tam, gdzie jest to możliwe.
- Wykorzystywanie kilku klas z biblioteki nie jest bezpieczne dla wątków.

Kolekcje bezpieczne dla wątków

Gdy Java była młodym językiem, Doug Lea napisał doskonałą książkę[8] *Concurrent Programming in Java*. W książce tej przedstawił proces tworzenia kilku kolekcji bezpiecznych dla wątków, które później stały się częścią JDK i zostały umieszczone w pakiecie `java.util.concurrent`. Kolekcje z tego pakietu są bezpieczne w przypadkach operacji wielowątkowych i działają odpowiednio szybko.

[8] [Lea99].

Faktycznie implementacja `ConcurrentHashMap` w niemal każdej sytuacji działa szybciej niż `HashMap`. Pozwala ona na jednoczesny zapis i odczyt oraz udostępnia metody wspierające często używane operacje złożone, które w pozostałych przypadkach nie są bezpieczne dla wątków. Jeżeli środowiskiem instalacyjnym jest Java 5, należy korzystać z `ConcurrentHashMap`.

Dostępnych jest również kilka innych klas pozwalających na obsługę zaawansowanych projektów wielowątkowych. Kilka przykładów przedstawiono poniżej.

`ReentrantLock`	Blokada, która może być nałożona w jednej metodzie i zwolniona w innej.
`Semaphore`	Implementacja klasycznego semafora, czyli blokady z licznikiem.
`CountDownLatch`	Blokada, która oczekuje na określoną liczbę zdarzeń przed ponownym uruchomieniem wszystkich wątków oczekujących na niej. Pozwala ona, aby wszystkie wątki miały możliwość wystartowania w mniej więcej tym samym momencie.

Zalecenie: *Warto zapoznać się z dostępnymi klasami. W przypadku języka Java należy zapoznać się z `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`.*

Poznaj modele wykonania

Istnieje kilka różnych sposobów na partycjonowanie działania w aplikacjach współbieżnych. Przed ich przedstawieniem musimy zapoznać się z kilkoma podstawowymi definicjami.

Zasoby związane	Zasoby stałej wielkości i liczby używane w środowisku współbieżnym. Przykładem mogą być połączenia z bazą danych oraz bufory odczytu i zapisu o stałej wielkości.
Wzajemne wykluczanie	Tylko jeden wątek może odwoływać się do danych współużytkowanych w danym momencie.
Zagłodzenie	Jeden wątek z grupy wątków nie może działać przez zbyt długi czas lub ciągle. Na przykład dopuszczenie, aby szybko działające wątki zawsze pierwsze przechodziły przez blokadę, powoduje zagłodzenie wolniej działających wątków, jeżeli nie ma końca strumienia szybko działających wątków.
Zakleszczenie	Co najmniej dwa wątki czekają na zakończenie innego z oczekujących wątków. Każdy wątek posiada zasób, który jest wymagany przez inny wątek, i żaden nie może się zakończyć, dopóki nie uzyska innego zasobu.
Uwięzienie	Wątki są wstrzymane, ale każdy z nich próbuje znaleźć „inną drogę". Z powodu rezonansu wątki próbujące iść dalej nie są w stanie zrobić tego przez nadmiernie długi czas — lub też nigdy.

Mając te definicje, możemy przedstawić różne modele wykonania kodu używane w programowaniu współbieżnym.

Producent-konsument[9]

Co najmniej jeden wątek producent wykonuje pewne zadania i umieszcza wyniki w buforze lub kolejce. Co najmniej jeden wątek konsument odbiera te wyniki i wykonuje na nich operacje. Kolejka pomiędzy producentami i konsumentami jest *zasobem związanym*. Oznacza to, że producenci muszą

[9] *http://pl.wikipedia.org/wiki/Problem_producenta_i_konsumenta*

Więcej na: www.ebook4all.pl

czekać na wolne miejsce w kolejce przed zapisem, a konsumenci muszą czekać, aż w kolejce będzie coś do skonsumowania. Koordynacja pomiędzy producentami i konsumentami poprzez kolejkę wymaga zastosowania sygnalizacji przez producenta i konsumenta. Producent zapisuje do kolejki i sygnalizuje, że kolejka nie jest już pusta. Konsumenci odczytują z kolejki i sygnalizują, że kolejka nie jest już pełna. Obie strony oczekują na informację, że mogą kontynuować pracę.

Czytelnik-pisarz[10]

Gdy mamy współużytkowany zasób, który przede wszystkim służy jako źródło informacji dla czytelników, ale jest okazjonalnie aktualizowany przez pisarzy, problemem jest przepustowość. Położenie nacisku na przepustowość może powodować zagłodzenie i akumulację oczekujących informacji. Umożliwienie aktualizacji może zmniejszać przepustowość. Koordynacja czytelników, aby nie czytali czegoś, co jest aktualizowane przez pisarza, i odwrotnie, jest zadaniem trudnym do wyważenia. Pisarze zwykle blokują wielu czytelników przez długi czas, powodując problemy z przepustowością.

Wyzwaniem jest zbilansowanie potrzeb zarówno czytelników, jak i pisarzy, aby zapewnić prawidłowe działanie, rozsądną przepustowość i uniknąć zagłodzenia. Prosta strategia wymaga, aby pisarze przed wykonaniem aktualizacji czekali, dopóki nie będzie żadnego czytelnika. Jeżeli będzie występował stały strumień czytelników, pisarz zostanie zagłodzony. Z drugiej strony, jeżeli pisarze będą działali często i będą mieli duży priorytet, spadnie przepustowość. Rozwiązanie tego problemu polega na znalezieniu równowagi i uniknięciu problemów ze współbieżną aktualizacją.

Ucztujący filozofowie[11]

Wyobraźmy sobie kilku filozofów siedzących wokół okrągłego stołu. Po lewej stronie każdego filozofa znajduje się widelec. Na środku stołu znajduje się duża misa spaghetti. Filozofowie spędzają swój czas na myśleniu, o ile nie są głodni. Gdy są głodni, chwytają za widelce leżące po obu stronach i zaczynają jeść. Filozofowie nie mogą jeść, jeżeli nie trzymają dwóch widelców. Jeżeli filozof siedzący po prawej stronie używa jednego z potrzebnych widelców, głodny filozof musi poczekać, dopóki ten po prawej nie skończy jeść i nie odłoży widelców. Gdy filozof jest najedzony, odkłada swoje widelce na stół i czeka, aż znów będzie głodny.

Gdy zamienimy filozofów na wątki i widelce na zasoby, problem stanie się podobny do wielu aplikacji korporacyjnych, w których procesy rywalizują o zasoby. Jeżeli system taki nie będzie odpowiednio zaprojektowany, tego rodzaju rywalizacja będzie powodowała zakleszczenia, uwięzienia oraz spadek przepustowości i efektywności systemu.

Większość problemów współbieżnych, z jakimi się spotykamy, jest odmianą jednej z tych trzech sytuacji. Warto zapoznać się z tymi algorytmami i napisać ich rozwiązania, dzięki czemu, gdy napotkamy na konkretny problem współbieżności, będziemy lepiej przygotowani do jego rozwiązania.

Zalecenie: *Warto nauczyć się podstawowych algorytmów i zrozumieć ich rozwiązania.*

---

[10] *http://pl.wikipedia.org/wiki/Problem_czytelnik%C3%B3w_i_pisarzy*

[11] *http://pl.wikipedia.org/wiki/Problem_ucztuj%C4%85cych_filozof%C3%B3w*

Uwaga na zależności pomiędzy synchronizowanymi metodami

Zależności pomiędzy synchronizowanymi metodami powodują subtelne błędy w kodzie współbieżnym. Język Java posiada notację `synchronized`, która pozwala chronić poszczególne metody. Jeżeli jednak mamy więcej metod synchronizowanych w tej samej klasie współużytkowanej, to system może być napisany nieprawidłowo[12].

Zalecenie: *Unikaj używania więcej niż jednej metody w obiekcie współużytkowanym.*

Zdarzają się jednak przypadki, gdy konieczne jest użycie więcej niż jednej metody we współużytkowanym obiekcie. W takim przypadku istnieją trzy sposoby na zapewnienie prawidłowego działania kodu.

- **Blokowanie po stronie klienta** — klient blokuje serwer przed wywołaniem pierwszej metody i blokada ta jest utrzymywana przez czas wykonywania kodu ostatniej metody.
- **Blokowanie po stronie serwera** — wewnątrz serwera tworzona jest metoda, która blokuje serwer, wywołuje wszystkie potrzebne metody, a następnie zdejmuje blokadę. Klient musi wywołać nową metodę.
- **Serwer adaptujący** — tworzony jest pośrednik zapewniający blokowanie. Jest to przykład blokowania po stronie serwera, w którym oryginalny serwer nie może być zmieniany.

Tworzenie małych sekcji synchronizowanych

Słowo kluczowe `synchronized` wprowadza blokadę. Wszystkie sekcje kodu chronione przez tę samą blokadę mają zagwarantowane, że w danym momencie tylko jeden wątek będzie wykonywał chronioną sekcję. Blokady są kosztowne, ponieważ powodują opóźnienia i dodają narzut. Dlatego nie należy nadużywać instrukcji `synchronized`. Z drugiej strony, sekcje krytyczne[13] muszą być chronione. Dlatego powinniśmy projektować nasz kod z minimalną możliwą liczbą sekcji krytycznych.

Niektórzy naiwni programiści próbują to osiągnąć, tworząc bardzo duże sekcje krytyczne. Jednak rozciąganie synchronizacji poza minimalną sekcję krytyczną zwiększa rywalizację o zasoby i obniża wydajność systemu[14].

Zalecenie: *Sekcje synchronizowane powinny być jak najmniejsze.*

---

[12] Patrz „Zależności pomiędzy metodami mogą uszkodzić kod współbieżny" w dodatku A.

[13] Sekcja krytyczna jest dowolną sekcją kodu, która musi być chroniona przed jednoczesnym użyciem, aby działanie programu było prawidłowe.

[14] Patrz „Zwiększanie przepustowości" w dodatku A.

Pisanie prawidłowego kodu wyłączającego jest trudne

Konstruowanie systemów, które mają działać w nieskończoność, różni się od tworzenia czegoś, co działa przez chwilę, a następnie elegancko się wyłącza.

Eleganckie wyłączenie może być trudne do osiągnięcia. Często spotykanymi problemami są zakleszczenia[15] wątków oczekujących na sygnał kontynuowania, który nigdy nie nadejdzie.

Wyobraźmy sobie system z wątkiem głównym, który generuje kilka wątków potomnych i czeka na ich zakończenie, po czym zwalnia zasoby i kończy pracę. Co się stanie, gdy wątek potomny ulegnie zakleszczeniu? Wątek główny będzie oczekiwał bez przerwy i system nigdy nie zostanie wyłączony.

Weźmy też pod uwagę podobny system, który został *poinstruowany* o konieczności zakończenia pracy. Wątek główny powiadamia wszystkie wątki potomne, aby przerwały swoje zadania i zakończyły się. Co się stanie, jeżeli dwa z tych wątków potomnych działają jako para producent-konsument? Załóżmy, że producent otrzymał sygnał z wątku głównego i szybko zakończył pracę. Konsument może oczekiwać na komunikat od producenta i być zablokowany w stanie, w którym nie może odebrać sygnału wyłączenia. Wątek utknie, czekając na producenta, i nie będzie mógł zakończyć pracy, co uniemożliwi również zakończenie pracy wątku głównego.

Tego typu sytuacje nie są niczym szczególnym. Jeżeli więc musimy napisać kod współbieżny, który wymaga eleganckiego zakończenia, należy oczekiwać, że spędzimy sporo czasu na prawidłowym zrealizowaniu takiego zakończenia.

Zalecenie: *Należy stosować wczesne wyłączanie i wczesne uruchamianie. Zwykle zajmuje to więcej czasu, niż oczekujemy. Warto przejrzeć istniejące algorytmy, ponieważ prawdopodobnie jest to trudniejsze, niż myślisz.*

Testowanie kodu wątków

Udowadnianie, że kod jest prawidłowy, jest niepraktyczne. Testowanie nie gwarantuje poprawności. Jednak dobre testy pozwalają zminimalizować ryzyko. To wszystko jest prawdą w przypadku rozwiązań jednowątkowych. Gdy tylko istnieją co najmniej dwa wątki korzystające z tego samego kodu i pracujące na współużytkowanych danych, wszystko staje się znacznie bardziej skomplikowane.

Zalecenie: *Warto pisać testy, które mają możliwość ujawnienia problemów, a następnie należy je często uruchamiać, korzystając z różnych konfiguracji programowych i systemowych oraz różnego obciążenia. Jeżeli w jakichkolwiek okolicznościach test zawiedzie, należy zlokalizować błąd. Nie należy ignorować awarii tylko dlatego, że w następnym uruchomieniu test wykonał się prawidłowo.*

---

[15] Patrz „Zakleszczenia" w dodatku A.

Należy tu wziąć pod uwagę wiele elementów. Poniżej zamieszczone są bardziej precyzyjne zalecenia.

- Traktujemy przypadkowe awarie jako potencjalne problemy z wielowątkowością.
- Na początku uruchamiamy kod niekorzystający z wątków.
- Nasz kod wątków powinien dać się włączać.
- Nasz kod wątków powinien dać się dostrajać.
- Uruchamiamy więcej wątków, niż mamy do dyspozycji procesorów.
- Uruchamiamy testy na różnych platformach.
- Uzbrajamy nasz kod w elementy próbujące wywołać awarie i wymuszające awarie.

Traktujemy przypadkowe awarie jako potencjalne problemy z wielowątkowością

Kod wielowątkowy powoduje, że awarii ulegają elementy, które „po prostu nie mogą zawieść". Większość programistów nie ma intuicyjnego poczucia, w jaki sposób wątki współdziałają z pozostałą częścią kodu (w tym autorzy). Błędy w kodzie wątków mogą się ujawniać raz na tysiąc lub milion wykonań. Próby ich powtórzenia mogą być frustrujące. Często prowadzi to do tego, że programiści kwalifikują błąd jako efekt promieniowania kosmicznego, wybryku sprzętu lub innego rodzaju „sytuacji jednorazowej". Najlepiej założyć, że sytuacje jednorazowe nie istnieją. Im dłużej są one ignorowane, tym więcej kodu korzysta z potencjalnie nieprawidłowego podejścia.

Zalecenie: *Nie należy ignorować awarii systemu, kwalifikując je jako jednorazowe.*

Na początku uruchamiamy kod niekorzystający z wątków

Może to się wydawać oczywiste, ale warto to powtórzyć. Należy upewnić się, że kod działa poza środowiskiem wielowątkowym. Zwykle oznacza to, że tworzymy obiekty POJO, które będą wywoływane przez nasze wątki. Obiekty POJO nie mają informacji o wątkach i dzięki temu mogą być testowane poza środowiskiem wielowątkowym. Im więcej elementów systemu jest umieszczonych w tych obiektach, tym lepiej.

Zalecenie: *Nie warto w tym samym czasie szukać błędów w kodzie jednowątkowym i wielowątkowym. Należy sprawdzić, czy kod działa poza wątkiem.*

Nasz kod wątków powinien dać się włączać

Warto pisać kod wspierający współbieżność w taki sposób, że może być uruchomiony w kilku konfiguracjach:

- Jeden wątek, kilka wątków, różna liczba w zależności od sposobu wykonania.
- Kod wątków współdziałający zarówno z rzeczywistym, jak i testowym środowiskiem.

- Wykonanie w środowisku testowym działającym szybko, powoli lub ze zmienną szybkością.
- Konfigurowanie testów w taki sposób, aby można było określać liczbę iteracji.

Zalecenie: *Warto tak napisać kod wątków, aby był dowolnie dołączany, by można było go uruchamiać w różnych konfiguracjach.*

Nasz kod wątków powinien dać się dostrajać

Uzyskanie właściwej równowagi wątków zwykle wymaga wielu prób. Jak najwcześniej należy znaleźć sposoby na sterowanie wydajnością systemu w różnych konfiguracjach. Takim sposobem może być łatwa zmiana liczby wątków. Warto rozważyć możliwość zmiany ich liczby w czasie działania systemu. Przydatny może być również mechanizm automatycznego dostrajania w zależności od przepustowości i obciążenia systemu.

Uruchamiamy więcej wątków, niż mamy do dyspozycji procesorów

Gdy system przełącza się pomiędzy zadaniami, zdarzają się różne sytuacje. Aby wymusić przełączanie zadań, należy uruchomić więcej wątków, niż mamy do dyspozycji procesorów lub rdzeni. Im częściej są przełączane zadania, tym większe prawdopodobieństwo, że wykryjemy kod, w którym brakuje sekcji krytycznej, lub kod powodujący zakleszczenie.

Uruchamiamy testy na różnych platformach

W roku 2007 tworzyliśmy kurs programowania współbieżnego. Sam kurs zakładał wykorzystanie przede wszystkim systemu OS X. W czasie lekcji korzystaliśmy też z Windows XP działającego na maszynie wirtualnej. Testy pisane w celu pokazania sytuacji błędnych nie były tak często wykonywane nieprawidłowo w środowisku XP, jak w środowisku OS X.

We wszystkich przypadkach wiedzieliśmy, że testowany kod był nieprawidłowy. Ilustruje to fakt, że różne systemy operacyjne mają różne zasady wielowątkowości, które wpływają na wykonywanie kodu. Kod wielowątkowy działa inaczej w różnych środowiskach[16]. Powinniśmy wykonywać testy w każdym potencjalnym środowisku instalacyjnym.

Zalecenie: *Kod wielowątkowy powinniśmy uruchamiać na wszystkich docelowych platformach — i to często.*

[16] Czy wiesz, że model wielowątkowości w języku Java nie gwarantuje wielowątkowości z wywłaszczaniem? Nowoczesne systemy operacyjne obsługują wielowątkowość z wywłaszczaniem, więc mamy ją „za darmo". Pomimo tego nie jest ona gwarantowana przez JVM.

Uzbrajamy nasz kod w elementy próbujące wywołać awarie i wymuszające awarie

Nie jest niczym niezwykłym fakt, że w kodzie współbieżnym ukrywają się błędy. Proste testy po prostu nie są w stanie ich ujawnić. Często ukrywają się w czasie normalnego przetwarzania. Mogą się one ujawniać jednokrotnie co kilka godzin, dni lub tygodni!

Powodem tego, że błędy współbieżności mogą być rzadkie, sporadyczne i trudne do powtórzenia, jest to, że tylko kilka ścieżek wykonania z tysięcy możliwych ścieżek przez wrażliwą sekcję może zawieść. Dlatego prawdopodobieństwo wyboru niewłaściwej ścieżki może być bardzo niskie. Powoduje to, że wykrywanie błędów i debugowanie jest bardzo trudne.

Jak można zwiększyć szanse przechwycenia takich rzadkich przypadków? Można uzbroić nasz kod i wymusić działanie w różnej kolejności przez dodanie wywołań takich metod, jak `Object.wait()`, `Object.sleep()`, `Object.yield()` oraz `Object.priority()`.

Każda z tych metod może wpływać na kolejność wykonania, zwiększając możliwość wykrycia błędu. Będzie lepiej, gdy nieprawidłowy kod zawiedzie wcześniej i możliwie często.

Istnieją dwie możliwości instrumentacji kodu:

- ręczna,
- automatyczna.

Instrumentacja ręczna

Do naszego kodu możemy ręcznie wstawiać wywołania `wait()`, `sleep()`, `yield()` oraz `priority()`. Może to być właściwe działanie, gdy testujemy szczególnie oporny fragment kodu.

Poniżej mamy przykład takiego działania:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        String url = urlGenerator.next();
        Thread.yield(); // Wstawione do testowania.
        updateHasNext();
        return url;
    }
    return null;
}
```

Wstawione wywołanie `yield()` zmienia ścieżkę wywołania kodu i być może powoduje awarię kodu w miejscu, w którym wcześniej wykonywał się prawidłowo. Jeżeli kod ulegnie awarii, nie jest to spowodowane dodaniem wywołania `yield()`[17]. Kod już wcześniej zawierał błąd, a teraz po prostu go ujawniliśmy.

[17] Jednak nie do końca. Ponieważ JVM nie gwarantuje wielowątkowości z wywłaszczaniem, określony algorytm może zawsze działać w systemach niewywłaszczających wątków. Sytuacja odwrotna jest również możliwa, ale z innych powodów.

Z podejściem tym wiąże się kilka problemów:

- Konieczne jest ręczne znalezienie odpowiednich miejsc.
- Skąd będziemy wiedzieli, gdzie umieścić wywołanie i jakiego rodzaju?
- Pozostawienie takiego kodu w środowisku produkcyjnym niepotrzebnie spowalnia kod.
- Jest to metoda śrutówki. Możemy znaleźć błąd lub nie. I tak nie mamy wielkich szans.

Potrzebujemy sposobu na korzystanie z tego podejścia w czasie testowania, a nie produkcji. Potrzebujemy również sposobu na modyfikowanie konfiguracji pomiędzy uruchomieniami, co zwiększa szanse znalezienia błędów.

Jasne jest, że gdy podzielimy nasz system na obiekty POJO, które nie mają informacji o wątkach, i klasy sterujące wątkami, łatwiej będzie znaleźć odpowiednie miejsca instrumentacji kodu. Co więcej, możemy utworzyć wiele różnych elementów testowych, które wywołują POJO w różnych reżimach wywołań `sleep`, `yield` i innych.

Instrumentacja automatyczna

Możemy skorzystać z narzędzi takich jak Aspect-Oriented Framework, CGLIB lub ASM do programowej instrumentacji naszego kodu. Możemy na przykład skorzystać z klasy z jedną metodą:

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

Można dodać jej wywołania w różnych miejscach kodu:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJiglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJiglePoint.jiggle();
        updateHasNext();
        ThreadJiglePoint.jiggle();
        return url;
    }
    return null;
}
```

Teraz można użyć prostego aspektu, który losowo wybiera jedną z możliwości: wywołanie `sleep`, `yield` lub niewywoływanie niczego.

Można również skorzystać z klasy `ThreadJigglePoint` mającej dwie implementacje. Pierwsza implementacja `jigle` nic nie robi i jest używana w produkcji. Druga generuje liczbę losową wybierającą pomiędzy `sleep`, `yield` lub niewywoływaniem niczego. Jeżeli uruchomimy nasz test tysiąc razy z losowym wytrząsaniem, możemy ujawnić niektóre błędy. Jeżeli test zostanie wykonany prawidłowo, będziemy mogli powiedzieć, że przynajmniej wykazaliśmy się należytą starannością. Choć to rozwiązanie jest dosyć proste, może być rozsądną opcją w porównaniu z zastosowaniem bardziej skomplikowanych narzędzi.

Dostępny jest program ConTest[18] opracowany w IBM, który działa podobnie, ale jego użycie jest znacznie bardziej skomplikowane.

Naszym celem jest wstrząsanie kodem tak, aby wątki wykonywały się za każdym razem w innej kolejności. Kombinacja dobrze napisanych testów i wstrząsów może zdecydowanie zwiększyć szanse znalezienia błędów.

Zalecenie: *Warto korzystać ze strategii wstrząsania w celu upolowania błędów.*

Zakończenie

Trudno uzyskać prawidłowy kod współbieżny. Kod, który jest prosty do analizy, może stać się koszmarem, gdy wymieszamy go z wieloma wątkami i współużytkowanymi danymi. Jeżeli chcemy zmierzyć się z kodem współbieżnym, musimy rygorystycznie przestrzegać pisania czystego kodu, ponieważ w przeciwnym razie zderzymy się z subtelnymi i niezbyt częstymi awariami.

Po pierwsze i najważniejsze, należy stosować zasadę pojedynczej odpowiedzialności. Należy podzielić nasz system na obiekty POJO, które oddzielają kod obsługi wątków od kodu nieobsługującego wątków. Przy testowaniu kodu obsługi wątków należy upewnić się, że testujemy tylko ten problem. Wynika z tego, że kod obsługi wątków powinien być mały i ściśle ukierunkowany.

Znane są możliwe źródła problemów z wielowątkowością: wiele wątków działa na wspólnych danych lub korzysta ze wspólnej puli zasobów. Przypadki brzegowe, takie jak czyste wyłączenie lub zakończenie iteracji pętli, mogą być szczególnie kłopotliwe.

Warto zapoznać się z biblioteką i podstawowymi algorytmami. Konieczne jest zrozumienie, w jaki sposób funkcje oferowane przez bibliotekę wspierają rozwiązywanie problemów podobnych do podstawowych algorytmów.

Należy nauczyć się rozpoznawania regionów kodu, które muszą być odizolowane i zablokowane. Nie należy blokować regionów kodu, które nie muszą być zablokowane. Należy unikać wywoływania jednej blokowanej sekcji z innej. Wymaga to głębokiego zrozumienia tego, czy dany element jest, czy nie jest współużytkowany. Liczba współużytkowanych obiektów i zakres współużytkowania powinny być możliwie małe. Projekt obiektów zawierających współużytkowane dane powinien być zmieniony tak, aby przyjmować klienty, zamiast wymuszać na nich zarządzanie stanem współużytkowanym.

Na pewno będą ujawniały się problemy. Te problemy, które nie zostaną ujawnione odpowiednio wcześnie, często są interpretowane jako przypadki jednorazowe. Te tak zwane przypadki jednorazowe zwykle ujawniają się pod obciążeniem systemu lub — pozornie — losowo. Z tego powodu musimy zapewnić, by nasz kod obsługi wątków można było uruchamiać w wielu konfiguracjach oraz na

[18] *http://www.alphaworks.ibm.com/tech/contest*

wielu platformach w sposób ciągły i powtarzalny. Możliwość testowania, która wynika naturalnie z trzech praw TDD, zakłada pewien poziom dołączania i pozwala obsługiwać niezbędny kod uruchomieniowy w szerokim zakresie konfiguracji.

Znacznie poprawimy nasze szanse znalezienia nieprawidłowego kodu, jeżeli poświęcimy trochę czasu na jego instrumentację. Można to zrobić ręcznie lub skorzystać z technologii automatycznych. Warto w to zainwestować możliwie wcześnie. Nasz kod korzystający z wątków powinien — przed przekazaniem go do produkcji — działać tak długo, jak tylko to możliwe.

Bibliografia

[Lea99]: Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns*, wydanie drugie, Prentice Hall 1999.

[PPP]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall 2002.

[PRAG]: Andrew Hunt, Dave Thomas, *The Pragmatic Programmer*, Addison-Wesley 2000.

Udane oczyszczanie kodu

Analiza przypadku procesora argumentów wiersza polecenia

ROZDZIAŁ TEN JEST ANALIZĄ PRZYPADKU udanego oczyszczania kodu modułu, w którym wystąpiły problemy ze skalowaniem. Przedstawimy, w jaki sposób moduł ten został przebudowany i wyczyszczony.

Większość z nas od czasu do czasu musi analizować argumenty wiersza polecenia. Jeżeli nie mamy wygodnego narzędzia, to po prostu przeglądamy tablicę ciągów przekazaną do funkcji `main`. Dostępnych jest kilka niezłych narzędzi pochodzących z różnych źródeł, ale żadne nie oferowało wszystkich oczekiwanych przez nas możliwości. Z tego powodu zdecydowaliśmy się napisać własny moduł. Nazwałem go `Args`.

Jest on bardzo prosty w użyciu. Tworzymy klasę `Args` z argumentami wejściowymi oraz ciągiem formatującym, a następnie odczytujemy wartości argumentów z obiektu `Args`. Weźmy pod uwagę następujący prosty przykład:

LISTING 14.1. Proste zastosowanie klasy Args

```
public static void main(String[] args) {
  try {
    Args arg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    int port = arg.getInt('p');
    String directory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Błąd argumentów: %s\n", e.errorMessage());
  }
}
```

Jak widać, jest to bardzo proste. Tworzymy obiekt klasy `Args` z dwoma parametrami. Pierwszym jest ciąg formatu lub *schematu*: `"l,p#,d*"`. Definiuje on trzy argumenty wiersza polecenia. Pierwszy, `-l`, jest argumentem logicznym. Drugi, `-p`, jest argumentem całkowitym. Trzeci, `-d`, jest argumentem tekstowym. Drugim parametrem konstruktora `Args` jest po prostu tablica argumentów wiersza polecenia przekazana do `main`.

Jeżeli konstruktor wykona się bez zgłoszenia wyjątku `ArgsException`, to wiemy, że przekazany wiersz polecenia został zanalizowany i że można korzystać z obiektu `Args`. Metody `getBoolean`, `getInteger` i `getString` pozwalają na odczytywanie wartości argumentów przy użyciu ich nazw.

Jeżeli wystąpi problem w ciągu formatującym albo w samych argumentach wiersza polecenia, zgłaszany jest wyjątek `ArgsException`. Dokładny opis tego, co poszło źle, można odczytać za pomocą metody `errorMessage` z wyjątku.

Implementacja klasy Args

Na listingu 14.2 przedstawiona jest implementacja klasy `Args`. Proszę o jego uważne przeczytanie. Ciężko pracowałem nad stylem i strukturą, więc mam nadzieję, że jest warta emulowania.

LISTING 14.2. Args.java

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
  private Map<Character, ArgumentMarshaler> marshalers;
  private Set<Character> argsFound;
  private ListIterator<String> currentArgument;

  public Args(String schema, String[] args) throws ArgsException {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();
```

```
  parseSchema(schema);
  parseArgumentStrings(Arrays.asList(args));
}

private void parseSchema(String schema) throws ArgsException {
  for (String element : schema.split(","))
    if (element.length() > 0)
      parseSchemaElement(element.trim());
}

private void parseSchemaElement(String element) throws ArgsException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length() == 0)
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (elementTail.equals("*"))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (elementTail.equals("#"))
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  else if (elementTail.equals("##"))
    marshalers.put(elementId, new DoubleArgumentMarshaler());
  else if (elementTail.equals("[*]"))
    marshalers.put(elementId, new StringArrayArgumentMarshaler());
  else
    throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
}

private void validateSchemaElementId(char elementId) throws ArgsException {
  if (!Character.isLetter(elementId))
    throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
}

private void parseArgumentStrings(List<String> argsList) throws ArgsException
{
  for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
  {
    String argString = currentArgument.next();
    if (argString.startsWith("-")) {
      parseArgumentCharacters(argString.substring(1));
    } else {
      currentArgument.previous();
      break;
    }
  }
}

private void parseArgumentCharacters(String argChars) throws ArgsException {
  for (int i = 0; i < argChars.length(); i++)
    parseArgumentCharacter(argChars.charAt(i));
}

private void parseArgumentCharacter(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null) {
    throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
  } else {
    argsFound.add(argChar);
    try {
      m.set(currentArgument);
    } catch (ArgsException e) {
      e.setErrorArgumentId(argChar);
      throw e;
    }
```

```
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public int nextArgument() {
    return currentArgument.nextIndex();
  }

  public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String[] getStringArray(char arg) {
    return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
  }
}
```

Warto zauważyć, że można czytać ten kod od góry do dołu bez skakania po kodzie lub wyszukiwania jakichś elementów. Jedynym elementem, którego trzeba szukać, jest definicja `ArgumentMarshaler`, którą rozmyślnie pozostawiłem bez zmian. Po uważnym przeczytaniu kodu powinniśmy rozumieć, jakie jest przeznaczenie interfejsu `ArgumentMarshaler` i klasy, która go implementuje. Kilka z nich przedstawiono poniżej (listingi od 14.3 do 14.6).

LISTING 14.3. ArgumentMarshaler.java

```
public interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
}
```

LISTING 14.4. BooleanArgumentMarshaler.java

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
  }
}
```

LISTING 14.5. StringArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return "";
  }
}
```

LISTING 14.6. IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }

  public static int getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof IntegerArgumentMarshaler)
      return ((IntegerArgumentMarshaler) am).intValue;
    else
      return 0;
  }
}
```

Pozostałe interfejsy dziedziczące po `ArgumentMarshaler` dla typów `double` i tablic `String` po prostu powielają ten wzorzec. Pozostawię je do wykonania jako ćwiczenie.

Problematyczny może być jeszcze jeden fragment — definicja stałych kodów błędów. Znajdują się one w klasie ArgsException (listing 14.7).

LISTING 14.7. ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode,
                       char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() {
    switch (errorCode) {
      case OK:
        return "TILT: Niedostępne.";
      case UNEXPECTED_ARGUMENT:
      return String.format("Nieoczekiwany argument -%c.", errorArgumentId);
      case MISSING_STRING:
      return String.format("Nie można znaleźć parametru znakowego dla -%c.",
                           errorArgumentId);
      case INVALID_INTEGER:
```

```
        return String.format("Argument -%c oczekuje liczby całkowitej, a był '%s'.",
                             errorArgumentId, errorParameter);
        case MISSING_INTEGER:
          return String.format("Nie można znaleźć parametru całkowitego dla -%c.",
                               errorArgumentId);
        case INVALID_DOUBLE:
          return String.format("Argument -%c oczekuje liczby double, a był '%s'.",
                               errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
          return String.format("Nie można znaleźć parametru double dla -%c.",
                               errorArgumentId);
        case INVALID_ARGUMENT_NAME:
          return String.format("'%c' nie jest prawidłową nazwą argumentu.",
                               errorArgumentId);
        case INVALID_ARGUMENT_FORMAT:
          return String.format("'%s' nie jest prawidłowym formatem argumentu.",
                               errorParameter);
      }
      return "";
    }

    public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
  }
```

Jak widać, sporo kodu potrzeba do zrealizowania szczegółów tego prostego zagadnienia. Jednym z powodów jest zastosowanie dosyć rozwlekłego języka. Java, język o statycznych typach, wymaga użycia wielu słów w celu usatysfakcjonowania systemu typów. W językach takich jak Ruby, Python lub Smalltalk program ten byłby znacznie mniejszy[1].

Proszę o ponowne przeczytanie kodu. Szczególną uwagę proponuję zwrócić na sposób nazywania elementów programu, rozmiar funkcji oraz formatowanie kodu. Jeżeli Czytelnik jest doświadczonym programistą, może mieć uwagi do niektórych części stylu i struktury. Jednak można powiedzieć, że jako całość program ten jest ładnie napisany i ma czystą strukturę.

Na przykład powinno być oczywiste, w jaki sposób możemy dodać nowy typ argumentu, np. argument daty lub argument liczby zespolonej, a jego dodanie powinno zająć niewiele czasu i wysiłku. Mówiąc krótko, wymagałoby to napisania nowej klasy implementującej `ArgumentMarshaler` oraz nowej funkcji `getXXX` i nowego elementu `case` w funkcji `parseSchemaElement`. Prawdopodobnie konieczne będzie również dodanie nowego `ArgsException.ErrorCode` i nowego komunikatu błędu.

Jak to napisałem?

Proszę teraz o szczególną uwagę. Nie napisałem tego programu od początku do końca w jego obecnej postaci. Co ważniejsze, nie oczekuję, że ktokolwiek będzie w stanie pisać czyste i eleganckie programy w jednym przebiegu. Jeżeli nauczyliśmy się czegokolwiek w kilku ostatnich dekadach, to przede wszystkim tego, że programowanie jest bardziej rzemiosłem niż sztuką. Aby pisać czysty kod, musimy na początku napisać brudny kod, a *następnie go oczyścić*.

---

[1] Ostatnio przepisałem ten moduł na Ruby. Miał on około 1/7 wielkości przedstawionego tu modułu i nieco lepszą strukturę.

Nie powinno to być dla nas niespodzianką. Nauczyliśmy się tej prawdy w szkole podstawowej, gdy nasi nauczyciele próbowali (zwykle w męczarniach) nauczyć nas pisania szkiców prac. Uczyli nas procesu, w którym powinniśmy napisać zgrubny szkic, następnie drugi szkic, a potem kilka kolejnych szkiców, aż otrzymamy ostateczną wersję. Próbowali nam w ten sposób pokazać, że pisanie czystych tekstów to długi proces wielu poprawek.

Większość młodych programistów (podobnie jak większość absolwentów) nie stosuje się do tej porady. Uważają oni, że ich podstawowym celem jest napisanie działającego programu. Gdy „już działa", przechodzą do następnego zadania, pozostawiając ten „działający" program w dowolnym stanie, który doprowadził do jego „działania". Większość doświadczonych programistów wie, że jest to zawodowe samobójstwo.

Args — zgrubny szkic

Na listingu 14.8 przedstawiona jest wcześniejsza wersja klasy `Args`. Ona „działa". Jest jednak mocno zagmatwana.

LISTING 14.8. Args.java (pierwszy szkic)

```
import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs = new HashMap<Character, String>();
  private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
```

```
private boolean parseSchema() throws ParseException {
  for (String element : schema.split(",")) {
    if (element.length() > 0) {
      String trimmedElement = element.trim();
      parseSchemaElement(trimmedElement);
    }
  }
  return true;
}

private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    parseBooleanSchemaElement(elementId);
  else if (isStringSchemaElement(elementTail))
    parseStringSchemaElement(elementId);
  else if (isIntegerSchemaElement(elementTail)) {
    parseIntegerSchemaElement(elementId);
  } else {
    throw new ParseException(
      String.format("Argument: %c ma niewłaściwy format: %s.",
                    elementId, elementTail), 0);
  }
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Zły znak:" + elementId + "w formacie Args: " + schema, 0);
  }
}

private void parseBooleanSchemaElement(char elementId) {
  booleanArgs.put(elementId, false);
}

private void parseIntegerSchemaElement(char elementId) {
  intArgs.put(elementId, 0);
}

private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, "");
}

private boolean isStringSchemaElement(String elementTail) {
  return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
  return elementTail.length() == 0;
}

private boolean isIntegerSchemaElement(String elementTail) {
  return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
  for (currentArgument = 0; currentArgument < args.length; currentArgument++)
  {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
```

```
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    if (isBooleanArg(argChar))
      setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
      setStringArg(argChar);
    else if (isIntArg(argChar))
      setIntArg(argChar);
    else
      return false;
    return true;
  }

  private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
```

```
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private boolean isStringArg(char argChar) {
  return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Niedostępne.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Nie można znaleźć parametru znakowego dla -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c oczekuje liczby całkowitej, a był '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Nie można znaleźć parametru całkowitego dla -%c.",
                           errorArgumentId);
  }
  return "";
}

private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(y) -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append(" nieoczekiwany.");
  return message.toString();
}

private boolean falseIfNull(Boolean b) {
  return b != null && b;
}

private int zeroIfNull(Integer i) {
  return i == null ? 0 : i;
}
```

```
  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }

  public int getInt(char arg) {
    return zeroIfNull(intArgs.get(arg));
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }

  private class ArgsException extends Exception {
  }
}
```

Mam nadzieję, że początkową reakcją na tę masę kodu było: „Mam nadzieję, że nie zostawi tego w takiej postaci!". Jeżeli taka była Twoja reakcja, to pamiętaj, jak inne osoby reagują na kod, który pozostawisz w formie zgrubnego szkicu.

„Zgrubny szkic" to najłagodniejsze określenie tego typu kodu. Jest to po prostu praca w toku. Ogromna liczba zmiennych instancyjnych jest zniechęcająca. Dziwne napisy, takie jak `TILT`, obiekty `HashSet` oraz `TreeSet`, bloki `try-catch-catch` jeszcze bardziej pogarszają sprawę.

Nie chciałem pisać takiego stosu śmieci. W rzeczywistości starałem się utrzymać rozsądną organizację. Można się o tym przekonać na podstawie wyboru nazw funkcji i zmiennych oraz na podstawie tego, że istnieje zgrubna struktura programu. Jednak jasne jest, że problem mi uciekł.

Bałagan stale przyrastał. Wcześniejsze wersje nie wyglądały tak źle. Na listingu 14.9 przedstawiona jest wcześniejsza wersja, w której działały tylko argumenty `Boolean`.

LISTING 14.9. Args.java (tylko Boolean)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;
```

```
public Args(String schema, String[] args) {
  this.schema = schema;
  this.args = args;
  valid = parse();
}

public boolean isValid() {
  return valid;
}

private boolean parse() {
  if (schema.length() == 0 && args.length == 0)
    return true;
  parseSchema();
  parseArguments();
  return unexpectedArguments.size() == 0;
}

private boolean parseSchema() {
  for (String element : schema.split(",")) {
    parseSchemaElement(element);
  }
  return true;
}

private void parseSchemaElement(String element) {
  if (element.length() == 1) {
    parseBooleanSchemaElement(element);
  }
}

private void parseBooleanSchemaElement(String element) {
  char c = element.charAt(0);
  if (Character.isLetter(c)) {
    booleanArgs.put(c, false);
  }
}

private boolean parseArguments() {
  for (String arg : args)
    parseArgument(arg);
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
  if (isBoolean(argChar)) {
    numberOfArguments++;
    setBooleanArg(argChar, true);
  } else
    unexpectedArguments.add(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}
```

```
  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return numberOfArguments;
  }

  public String usage() {
    if (schema.length() > 0)
      return "-["+schema+"]";
    else
      return "";
  }

  public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(y) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" nieoczekiwany.");
    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
  }
}
```

Choć można w tym kodzie znaleźć wiele miejsc wymagających zmiany, w całości nie wygląda tak źle. Jest zwarty, prosty i łatwy do zrozumienia. Jednak można tu łatwo zauważyć zalążki późniejszej sterty śmieci. Dosyć jasno widać, w jaki sposób rozrósł się późniejszy bałagan.

Warto zauważyć, że w późniejszym bałaganie mamy tylko o dwa typy argumentów więcej: `String` oraz `integer`. Dodanie tylko dwóch typów miało negatywny wpływ na kod. Coś, co było możliwe do utrzymania, zmieniło się w coś, co byłoby wypełnione błędami.

Dodałem tylko dwa typy argumentów. Na początku dodałem argument `String`, co dało w efekcie następujący kod:

LISTING 14.10. Args.java (Boolean oraz String)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
```

```
private Set<Character> unexpectedArguments = new TreeSet<Character>();
private Map<Character, Boolean> booleanArgs =
  new HashMap<Character, Boolean>();
private Map<Character, String> stringArgs =
  new HashMap<Character, String>();
private Set<Character> argsFound = new HashSet<Character>();
private int currentArgument;
private char errorArgument = '\0';

enum ErrorCode {
  OK, MISSING_STRING}

private ErrorCode errorCode = ErrorCode.OK;

public Args(String schema, String[] args) throws ParseException {
  this.schema = schema;
  this.args = args;
  valid = parse();
}

private boolean parse() throws ParseException {
  if (schema.length() == 0 && args.length == 0)
    return true;
  parseSchema();
  parseArguments();
  return valid;
}

private boolean parseSchema() throws ParseException {
  for (String element : schema.split(",")) {
    if (element.length() > 0) {
      String trimmedElement = element.trim();
      parseSchemaElement(trimmedElement);
    }
  }
  return true;
}

private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    parseBooleanSchemaElement(elementId);
  else if (isStringSchemaElement(elementTail))
    parseStringSchemaElement(elementId);
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Zły znak:" + elementId + "w formacie Args: " + schema, 0);
  }
}

private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, "");
}

private boolean isStringSchemaElement(String elementTail) {
  return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
```

```
    return elementTail.length() == 0;
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      valid = false;
    }
  }

  private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
      setBooleanArg(argChar, true);
    else if (isString(argChar))
      setStringArg(argChar, "");
    else
      set = false;
    return set;
  }

  private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {
      stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgument = argChar;
      errorCode = ErrorCode.MISSING_STRING;
    }
  }

  private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }
```

```
  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      switch (errorCode) {
        case MISSING_STRING:
          return String.format("Nie można znaleźć parametru znakowego dla -%c.",
                               errorArgument);
        case OK:
          throw new Exception("TILT: Niedostępne.");
      }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(y) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" nieoczekiwany.");
    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }

  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }
}
```

Jak widać, kod zaczął się wymykać spod kontroli. Nadal nie jest zły, ale jasne jest, że bałagan zaczął narastać. Jest to już stos, ale jeszcze niewypełniony śmieciami. Wystarczyło dodanie obsługi argumentów typu `integer`, aby stos zaczął ropieć i fermentować.

Zatrzymałem się

Miałem do dodania co najmniej dwa kolejne typy argumentów i od razu mogłem stwierdzić, że sprawy będą miały się jeszcze gorzej. Jeżeli bez zastanowienia parłbym naprzód, prawdopodobnie byłbym w stanie doprowadzić ten kod do prawidłowego działania, ale pozostawiając za sobą zbyt duży bałagan, aby można było go łatwo poprawić. Jeżeli struktura tego kodu miała być kiedykolwiek możliwa do utrzymania, był to najwyższy czas, aby ją poprawić.

Dlatego wstrzymałem dodawanie funkcji i zacząłem przebudowę. Po dodaniu obsługi argumentów typu `String` i `integer` wiedziałem już, że każdy typ argumentu wymaga nowego kodu w trzech głównych miejscach. Po pierwsze, każdy typ argumentu wymaga pewnego sposobu analizy elementów schematu w celu wybrania obiektu `HashMap` dla danego typu. Następnie każdy typ argumentów musi być pobrany z ciągu wiersza polecenia i skonwertowany na jego właściwy typ. Na koniec każdy typ argumentów wymaga metody `getXXX`, która zwraca wywołującemu właściwy typ.

Wiele różnych typów i wszystkie mają podobne metody — dla mnie to brzmi jak klasa. W ten sposób powstała koncepcja klasy `ArgumentMarshaler`.

O przyrostowości

Jednym z najlepszych sposobów na zrujnowanie programu jest wprowadzenie do jego struktury ogromnych zmian w imię usprawnień. Niektóre programy nigdy nie podnoszą się po takich „usprawnieniach". Problem wynika z tego, że bardzo trudno uzyskać takie samo działanie, jak przed „usprawnieniami".

Aby tego uniknąć, korzystam z dyscypliny programowania sterowanego testami (TDD). Jedną z podstawowych doktryn tego podejścia jest zachowanie działania systemu przez cały czas. Inaczej mówiąc, przy użyciu TDD nie mogę wprowadzić zmian do systemu, które przerywają jego działanie. Każda wprowadzona zmiana musi zachować poprzednie działanie systemu.

Aby to osiągnąć, potrzebuję zbioru testów automatycznych, które mogę uruchomić w celu sprawdzenia, czy działanie systemu nie zmieniło się. Dla klasy `Args` napisałem zbiór testów jednostkowych i akceptacyjnych już w czasie budowania sterty śmieci. Testy jednostkowe były napisane w języku Java i administrowane przez `JUnit`. Testy akceptacyjne były napisane jako strony wiki w `FitNesse`. Mogłem uruchomić te testy w dowolnym momencie i jeżeli wykonywały się prawidłowo, byłem pewny, że system działa w zdefiniowany sposób.

Zacząłem więc wprowadzać sporo niewielkich zmian. Każda zmiana przybliżała strukturę systemu w kierunku koncepcji użycia `ArgumentMarshaler`. Jednak po wprowadzeniu każdej zmiany system stale działał. Pierwszą zmianą, jaką wykonałem, było wstawienie szkieletu `ArgumentMarshaller` na szczyt sterty śmieci (listing 14.11).

LISTING 14.11. ArgumentMarshaller dodany do Args.java

```
private class ArgumentMarshaler {
  private boolean booleanValue = false;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {return booleanValue;}
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  }
}
```

Jasne jest, że nie spowodowało to powstania żadnego błędu. Następnie wprowadziłem jak najprostszą modyfikację, która nie powinna spowodować dużych usterek w kodzie. Zmieniłem `HashMap` dla argumentów `Boolean`, aby przechowywał obiekty `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
  new HashMap<Character, ArgumentMarshaler>();
```

Spowodowało to błąd w kilku instrukcjach, które szybko poprawiłem.

```
...
  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());
  }
...
  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).setBoolean(value);
  }
...
  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg).getBoolean());
  }
```

Warto zauważyć, że zmiany te znajdują się dokładnie w tych obszarach, które wcześniej wymieniłem: metody `parse`, `set` oraz `get` dla danego typu argumentu. Niestety, pomimo tak niewielkich zmian niektóre testy przestały działać. Jeżeli spojrzymy uważnie na `getBoolean`, zauważymy, że jeżeli wywołamy ją z argumentem `'y'`, a nie będzie argumentu `y`, to `booleanArgs.get('y')` zwróci `null`, a funkcja zgłosi `NullPointerException`. Funkcja `falseIfNull` została użyta do ochrony przed taką sytuacją, ale wprowadzona przeze mnie zmiana spowodowała, że funkcja ta przestała być ważna.

Przyrostowość wymaga, abym szybko doprowadził do działania kodu, przed wprowadzeniem jakichkolwiek innych zmian. W rzeczywistości poprawka nie była taka trudna. Po prostu przesunąłem kontrolę wartości `null`. Nie było już wartości `boolean` będącej `null`, którą musiałem sprawdzać; był teraz obiekt `ArgumentMarshaller`.

Po pierwsze, usunąłem `falseIfNull`, zastępując ją funkcją `getBoolean`. Była teraz bezużyteczna, więc wyeliminowałem całą funkcję. Testy były przerywane w ten sam sposób, więc byłem pewny, że nie wprowadziłem żadnych nowych błędów.

```
public boolean getBoolean(char arg) {
  return booleanArgs.get(arg).getBoolean();
}
```

Następnie podzieliłem funkcję na dwa wiersze i umieściłem `ArgumentMarshaler` w osobnej zmiennej o nazwie `argumentMarshaler`. Musiałem zająć się jednak długą nazwą zmiennej; była ona mocno nadmiarowa i zaciemniała funkcję. Skróciłem ją więc do `am` [N5].

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am.getBoolean();
}
```

Następnie dodałem kod wykrywania wartości `null`.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && am.getBoolean();
}
```

Argumenty typu String

Dodanie argumentów `String` było bardzo podobne do dodawania argumentów `boolean`. Musiałem zmienić `HashMap` oraz doprowadzić do prawidłowego działania funkcji `parse`, `set` oraz `get`. Nie powinno tu być żadnych niespodzianek, być może poza umieszczeniem całej implementacji szeregowania w klasie bazowej `ArgumentMarshaler` zamiast rozproszenia jej w klasach pochodnych.

```
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
...
  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, new StringArgumentMarshaler());
  }
...
  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.get(argChar).setString(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
...
  public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : am.getString();
  }
...
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
```

```
    private String stringValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }
  }
```

Ponownie zmiany te były wprowadzane jedna po drugiej i w taki sposób, że testy stale działały, choć nie przechodziły. Gdy test nie przechodził, upewniałem się, że przed przejściem do następnej zmiany będzie on przechodził prawidłowo.

Teraz moje intencje powinny być już widoczne. Gdy obecne funkcje szeregowania znajdują się w klasie bazowej `ArgumentMarshaler`, zaczynam przesuwać je do klas pochodnych. Powoduje to, że wszystko działa, przy jednoczesnej stopniowej zmianie kształtu programu.

Oczywistym krokiem było przeniesienie funkcji dla argumentu `int` do `ArgumentMarshaler`. I tym razem nie było żadnych niespodzianek.

```
private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
...
  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, new IntegerArgumentMarshaler());
  }
...
  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }
...
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
```

```
    }
...
    private class ArgumentMarshaler {
      private boolean booleanValue = false;
      private String stringValue;
      private int integerValue;

      public void setBoolean(boolean value) {
        booleanValue = value;
      }

      public boolean getBoolean() {
        return booleanValue;
      }

      public void setString(String s) {
        stringValue = s;
      }

      public String getString() {
        return stringValue == null ? "" : stringValue;
      }

      public void setInteger(int i) {
        integerValue = i;
      }

      public int getInteger() {
        return integerValue;
      }
    }
```

Gdy wszystkie funkcje szeregowania znalazły się w `ArgumentMarshaler`, zacząłem przenosić odpowiednie funkcje do jego klas pochodnych. Pierwszym krokiem było przeniesienie funkcji `setBoolean` do `BooleanArgumentMarshaller` i upewnienie się, że jest wywoływana prawidłowo. Utworzyłem więc abstrakcyjną metodę `set`.

```
  private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }
```

```
  public int getInteger() {
    return integerValue;
  }

  public abstract void set(String s);
}
```

Następnie zaimplementowałem metodę set w `BooleanArgumentMarshaller`.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  public void set(String s) {
    booleanValue = true;
  }
}
```

Na koniec zastąpiłem wywołanie `setBoolean` wywołaniem `set`.

```
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.get(argChar).set("true");
}
```

Testy nadal były wykonywane prawidłowo. Ponieważ zmiana ta spowodowała, że `set` została przeniesiona do `BooleanArgumentMarshaler`, usunąłem metodę `setBoolean` z klasy bazowej `ArgumentMarshaler`.

Należy zwrócić uwagę, że abstrakcyjna funkcja `set` oczekuje argumentu `String`, ale implementacja w `BooleanArgumentMarshaller` nie korzysta z niego. Umieściłem tam ten argument, ponieważ wiedziałem, że `StringArgumentMarshaller` oraz `IntegerArgumentMarshaller` *będą* z niego korzystały.

Następnie chciałem przenieść metodę `get` do `BooleanArgumentMarshaler`. Przenoszenie funkcji `get` jest zawsze mało eleganckie, ponieważ zwracanym typem zawsze musi być `Object` i w tym przypadku jest on rzutowany na `Boolean`.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean)am.get();
}
```

Aby kod się kompilował, dodałem funkcję `get` do `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
...
  public Object get() {
    return null;
  }
}
```

Kod ten kompiluje się i oczywiście nie przechodzi testów. Aby testy znów przechodziły, wystarczyło utworzyć abstrakcyjną metodę `get` i zaimplementować ją w `BooleanAgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
  protected boolean booleanValue = false;
...

  public abstract Object get();
}
```

```
    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
      public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }
```

Kolejny raz testy działają prawidłowo. Tak więc `get` i `set` są przeniesione do `BooleanArgument↪Marshaler`! Pozwoliło mi to usunąć starą funkcję `getBoolean` z `ArgumentMarshaler`, przenieść chronioną zmienną `booleanValue` do `BooleanArgumentMarshaler` i zmienić jej widoczność na `private`.

Ten sam szablon zmian zastosowałem dla typu `String`. Przeniosłem funkcje `set` i `get`, usunąłem nieużywane funkcje i przeniosłem zmienne.

```
    private void setStringArg(char argChar) throws ArgsException {
      currentArgument++;
      try {
        stringArgs.get(argChar).set(args[currentArgument]);
      } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }
...
    public String getString(char arg) {
      Args.ArgumentMarshaler am = stringArgs.get(arg);
      return am == null ? "" : (String) am.get();
    }
...
    private abstract class ArgumentMarshaler {
      private int integerValue;

      public void setInteger(int i) {
        integerValue = i;
      }

    public int getInteger() {
      return integerValue;
    }

    public abstract void set(String s);

    public abstract Object get();
  }

    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
      private boolean booleanValue = false;

      public void set(String s) {
        booleanValue = true;
      }

      public Object get() {
        return booleanValue;
      }
    }
```

```
  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
    }

    public Object get() {
      return null;
    }
  }
}
```

Na koniec powtórzyłem proces dla argumentów `integer`. Było to nieco bardziej skomplikowane, ponieważ liczby całkowite muszą być analizowane, a operacja `parse` może zgłosić wyjątek. Jednak wynik jest lepszy, ponieważ cała obsługa `NumberFormatException` została umieszczona w `IntegerArgumentMarshaler`.

```
private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }
...
  private void setBooleanArg(char argChar) {
    try {
      booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
...
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
  }
...
  private abstract class ArgumentMarshaler {
```

```
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }
...
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Oczywiście, testy stale były realizowane prawidłowo. Następnie zająłem się usunięciem różnych odwzorowań z algorytmu. Dzięki temu cały system stał się bardziej ogólny. Jednak nie mogłem ich usunąć przez proste skasowanie, ponieważ system przestałby działać. Zamiast tego dodałem nową zmienną `Map` do klasy `ArgumentMarshaler` i jedna po drugiej zmieniałem metody, aby korzystały z niej zamiast z oryginalnych zmiennych.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseIntegerSchemaElement(char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseStringSchemaElement(char elementId) {
    ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

Oczywiście, testy nadal były wykonywane prawidłowo. Następnie zmieniłem `isBooleanArg` z takiej postaci:

```
private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}
```

na taką:

```
private boolean isBooleanArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof BooleanArgumentMarshaler;
}
```

Testy nadal były wykonywane prawidłowo. Tak więc wprowadziłem tę samą zmianę do `isIntArg` oraz `isStringArg`.

```
private boolean isIntArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof StringArgumentMarshaler;
}
```

Testy nadal były wykonywane prawidłowo. Dzięki temu wyeliminowałem powielone wywołania `marshalers.get`:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (isBooleanArg(m))
    setBooleanArg(argChar);
  else if (isStringArg(m))
    setStringArg(argChar);
  else if (isIntArg(m))
    setIntArg(argChar);
  else
    return false;

  return true;
}
private boolean isIntArg(ArgumentMarshaler m) {
  return m instanceof IntegerArgumentMarshaler;
}
private boolean isStringArg(ArgumentMarshaler m) {
  return m instanceof StringArgumentMarshaler;
}
private boolean isBooleanArg(ArgumentMarshaler m) {
  return m instanceof BooleanArgumentMarshaler;
}
```

Sprawiło to, że nie było powodu istnienia metod `isxxxArgs`. Więc wbudowałem je:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(argChar);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;
  return true;
}
```

Następnie zacząłem korzystać ze zmiennych `marshalers` w funkcjach `set`, usuwając jednocześnie kontenery `Map`. Zacząłem od argumentów logicznych.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(m);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
...
private void setBooleanArg(ArgumentMarshaler m) {
  try {
    m.set("true"); // Było: booleanArgs.get(argChar).set("true");
  } catch (ArgsException e) {
  }
}
```

Testy nadal działały prawidłowo, więc to samo zrobiłem z typami `String` i `integer`. Pozwoliło mi to zintegrować część nieeleganckiego kodu zarządzania wyjątkami w funkcji `setArgument`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    m.set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}
```

```
private void setStringArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  try {
    m.set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
```

Byłem bliski usunięcia starych zmiennych `Map`. Musiałem jeszcze zmienić funkcję `getBoolean` z postaci:

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean) am.get();
}
```

na taką:

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}
```

Ta ostatnia zmiana mogła być niespodzianką. Dlaczego nagle zdecydowałem się obsługiwać `ClassCastException`? Powodem było to, że miałem zestaw testów jednostkowych i osobne testy akceptacyjne w FitNesse. Okazało się, że testy FitNesse sprawdzają, iż gdy wywołamy `getBoolean` na argumencie innym niż logiczny, otrzymamy wartość `false`. Testy jednostkowe tego nie robiły. Do tego momentu uruchamiałem wyłącznie testy jednostkowe[2].

Ta ostatnia zmiana pozwoliła mi wyodrębnić ostatnie zastosowanie mapy `boolean`:

```
private void parseBooleanSchemaElement(char elementId) {
  ArgumentMarshaler m = new BooleanArgumentMarshaler();
  booleanArgs.put(elementId, m);

  marshalers.put(elementId, m);
}
```

Teraz mogłem usunąć zmienną `Map` dla typu `boolean`.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
      new HashMap<Character, ArgumentMarshaler>();

  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
```

[2] Aby zapobiec tego typu niespodziankom, dodałem nowy test jednostkowy wywołujący wszystkie testy FitNesse.

```
    private Map<Character, ArgumentMarshaler> marshalers =
      new HashMap<Character, ArgumentMarshaler>();
...
```

Następnie przeniosłem argumenty `String` i `integer` w ten sam sposób i nieco wyczyściłem kod obsługi argumentów logicznych.

```
    private void parseBooleanSchemaElement(char elementId) {
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    }

    private void parseIntegerSchemaElement(char elementId) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    }

    private void parseStringSchemaElement(char elementId) {
      marshalers.put(elementId, new StringArgumentMarshaler());
    }
...
    public String getString(char arg) {
      Args.ArgumentMarshaler am = marshalers.get(arg);
      try {
        return am == null ? "" : (String) am.get();
      } catch (ClassCastException e) {
      return "";
      }
    }

  public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }
  ...
  public class Args {
  ...
  - private Map<Character, ArgumentMarshaler> stringArgs =
      new HashMap<Character, ArgumentMarshaler>();
  - private Map<Character, ArgumentMarshaler> intArgs =
      new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
      new HashMap<Character, ArgumentMarshaler>();
  ...
```

Wbudowałem trzy metody `parse`, ponieważ nie realizowały już zbyt wielu operacji:

```
  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
      throw new ParseException(String.format(
        "Argument: %c ma niewłaściwy format: %s.", elementId, elementTail), 0);
    }
  }
```

Spójrzmy teraz, jak wygląda to w całości. Na listingu 14.12 przedstawiona jest bieżąca postać klasy `Args`.

LISTING 14.12. Args.java (po pierwszej przebudowie)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
```

```
      throw new ParseException(String.format(
      "Argument: %c ma niewłaściwy format: %s.", elementId, elementTail), 0);
    }
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
        "Zły znak:" + elementId + "w formacie Args: " + schema, 0);
    }
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }

  private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
```

```
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    m.set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  try {
    m.set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private void setBooleanArg(ArgumentMarshaler m) {
  try {
    m.set("true");
  } catch (ArgsException e) {
  }
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Niedostępne.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Nie można znaleźć parametru znakowego dla -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c oczekuje liczby całkowitej, a był '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
```

```
        return String.format("Nie można znaleźć parametru całkowitego dla -%c.",
                             errorArgumentId);
      }
      return "";
    }

    private String unexpectedArgumentMessage() {
      StringBuffer message = new StringBuffer("Argument(y) -");
      for (char c : unexpectedArguments) {
        message.append(c);
      }
      message.append(" nieoczekiwany.");
      return message.toString();
    }

    public boolean getBoolean(char arg) {
      Args.ArgumentMarshaler am = marshalers.get(arg);
        boolean b = false;
      try {
        b = am != null && (Boolean) am.get();
      } catch (ClassCastException e) {
        b = false;
      }
      return b;
    }

    public String getString(char arg) {
        Args.ArgumentMarshaler am = marshalers.get(arg);
      try {
        return am == null ? "" : (String) am.get();
      } catch (ClassCastException e) {
        return "";
      }
    }

    public int getInt(char arg) {
      Args.ArgumentMarshaler am = marshalers.get(arg);
      try {
        return am == null ? 0 : (Integer) am.get();
      } catch (Exception e) {
        return 0;
      }
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public boolean isValid() {
      return valid;
    }

    private class ArgsException extends Exception {
    }
      private abstract class ArgumentMarshaler {
      public abstract void set(String s) throws ArgsException;
      public abstract Object get();
    }

    private class BooleanArgumentMarshaler extends ArgumentMarshaler {
      private boolean booleanValue = false;

      public void set(String s) {
        booleanValue = true;
      }
```

```
      public Object get() {
        return booleanValue;
      }
    }

    private class StringArgumentMarshaler extends ArgumentMarshaler {
      private String stringValue = "";

      public void set(String s) {
        stringValue = s;
      }
      public Object get() {
        return stringValue;
      }
    }

    private class IntegerArgumentMarshaler extends ArgumentMarshaler {
      private int intValue = 0;

      public void set(String s) throws ArgsException {
        try {
          intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
          throw new ArgsException();
        }
      }

      public Object get() {
        return intValue;
      }
    }
  }
```

Po tak dużej ilości pracy efekt trochę rozczarowuje. Struktura jest nieco lepsza, ale nadal mamy te wszystkie zmienne na górze, nadal w `setArgument` znajduje się okropna konstrukcja `case`, a wszystkie funkcje `set` są naprawdę brzydkie. Nie wspominam tu nawet o przetwarzaniu błędów. Przed nami wciąż sporo pracy.

Naprawdę chciałem usunąć instrukcje `case` zależne od typu z `setArgument` [G23]. Chciałbym, aby `setArgument` był jedynym wywołaniem `ArgumentMarshaler.set`. Oznacza to, że powinienem przenieść `setIntArg`, `setStringArg` oraz `setBooleanArg` do odpowiednich klas dziedziczących po `ArgumentMarshaler`. Ale wtedy pojawił się problem.

Jeżeli spojrzymy na `setIntArg`, okaże się, że korzysta z dwóch zmiennych instancyjnych: `args` oraz `currentArg`. Aby przenieść `setIntArg` do `BooleanArgumentMarshaler`, musiałbym przekazywać zarówno `args`, jak i `currentArgs` jako argumenty funkcji. Jest to nieeleganckie rozwiązanie [F1]. Wolę przekazywać jeden argument zamiast dwóch. Na szczęście istnieje proste rozwiązanie. Możemy skonwertować tablicę `args` na `list` i przekazać do funkcji `set` jej `Iterator`. Poniższy kod wykonuje dziesięć kroków, pozwalających na spełnienie testu po teście. Zamieszczę tu tylko ostateczny wynik. Czytelnik powinien zorientować się, jakie były poszczególne małe kroki.

```
public class Args {
  private String schema;
  - private String[] args;

  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
```

```
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
  private List<String> argsList;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
---
  private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }
    return true;
  }
---
  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      m.set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
      m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
}
```

Były to proste zmiany, dzięki którym testy nadal działały. Teraz zacząłem przenosić funkcje `set` do odpowiednich klas podrzędnych. Na początek konieczne było wprowadzenie następującej zmiany w `setArgument`:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

Zmiana ta była istotna, ponieważ zamierzałem całkowicie wyeliminować łańcuch `if-else`. Dlatego chciałem wyodrębnić z niego kontrolę błędu.

Teraz możemy rozpocząć przenoszenie funkcji `set`. Funkcja `setBooleanArg` jest najprostsza, więc zaczniemy od niej. Naszym celem jest zmiana `setBooleanArg` na zwykłe przekazanie wywołania do `BooleanArgumentMarshaler`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m, currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
---
private void setBooleanArg(ArgumentMarshaler m,
                           Iterator<String> currentArgument)
                           throws ArgsException {
  try {
    m.set("true");
  } catch (ArgsException e) {
  }
}
```

Czy nie dodawaliśmy wcześniej tego przetwarzania wyjątku? Dodawanie elementów, aby je później usunąć, jest dosyć częste w procesie przebudowywania. Wykonywanie niewielkich kroków oraz konieczność zachowania działania testów powodują, że często przesuwa się fragmenty kodu. Przebudowa podobna jest do układania kostki Rubika. Do osiągnięcia większego celu używa się wielu małych kroków. Każdy krok pozwala na następny.

Dlaczego przekazywaliśmy `iterator`, gdy `setBooleanArg` wcale go nie potrzebuje? Ponieważ `setIntArg` oraz `setStringArg` będą go potrzebowały! Oprócz tego, ponieważ chciałem uruchamiać te funkcje przez metodę abstrakcyjną w `ArgumentMarshaler`, musiałem przekazać ją do `setBooleanArg`.

Obecnie `setBooleanArg` jest bezużyteczna. Jeżeli mielibyśmy funkcję `set` w `ArgumentMarshaler`, moglibyśmy wywoływać ją bezpośrednio. Czas więc napisać tę funkcję! Pierwszym krokiem jest napisanie nowej metody abstrakcyjnej w `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
  public abstract void set(Iterator<String> currentArgument)
                           throws ArgsException;
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}
```

Oczywiście, spowoduje to, że wszystkie klasy pochodne przestaną działać. Dlatego konieczne będzie dodanie w każdej z nich nowej metody.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public void set(String s) {
    booleanValue = true;

}

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;
```

```
    public void set(Iterator<String> currentArgument) throws ArgsException {
    }

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
  }

    public Object get() {
      return intValue;
    }
  }
```

Teraz możemy wyeliminować `setBooleanArg`!

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

Wszystkie testy wykonują się prawidłowo, a funkcja `set` została przeniesiona do `BooleanArgument` ↪`Marshaler`! Teraz możemy wykonać to samo dla typu `String` i `integer`.

```
    private boolean setArgument(char argChar) throws ArgsException {
      ArgumentMarshaler m = marshalers.get(argChar);
      if (m == null)
        return false;
      try {
        if (m instanceof BooleanArgumentMarshaler)
          m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
          m.set(currentArgument);
        else if (m instanceof IntegerArgumentMarshaler)
          m.set(currentArgument);
      } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
      }
      return true;
    }
  ---
    private class StringArgumentMarshaler extends ArgumentMarshaler {
      private String stringValue = "";

      public void set(Iterator<String> currentArgument) throws ArgsException {
```

```
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }

    public void set(String s) {
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        set(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
      }
    }

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

I teraz *coup de grace* — wybór w zależności od typu może być usunięty! Touché!

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      m.set(currentArgument);
      return true;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
  }
```

Możemy już usunąć część niepotrzebnych funkcji w `IntegerArgumentMarshaler` i nieco posprzątać.

```
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

Możemy również przekształcić `ArgumentMarshaler` w interfejs.

```
private interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
  Object get();
}
```

Teraz zobaczymy, jak łatwo można dodać nowy typ argumentu do naszej struktury. Powinno to wymagać niewielu zmian, które to zmiany powinny być izolowane. Na początek zacznijmy od dodania nowego przypadku testowego, który sprawdza, czy argument `double` działa prawidłowo.

```
public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[] {"-x","42.3"});
  assertTrue(args.isValid());
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}
```

Teraz wyczyścimy kod analizy schematu i dodamy operację wykrywania ## dla argumentów typu `double`.

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length() == 0)
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (elementTail.equals("*"))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (elementTail.equals("#"))
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  else if (elementTail.equals("##"))
    marshalers.put(elementId, new DoubleArgumentMarshaler());
  else
```

```
      throw new ParseException(String.format(
        "Argument: %c ma niewłaściwy format: %s.", elementId, elementTail), 0);
  }
```

Następnie napiszemy klasę `DoubleArgumentMarshaler`.

```
  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
```

Wymusza to na nas dodanie nowej wartości `ErrorCode`.

```
  private enum ErrorCode {
      OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
      MISSING_DOUBLE, INVALID_DOUBLE}
```

Potrzebujemy również funkcji `getDouble`.

```
  public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }
```

Wszystkie testy są wykonywane prawidłowo! Poszło całkiem bezboleśnie. Teraz możemy się upewnić, że kod przetwarzania błędów działa prawidłowo. Następny przypadek testowy kontroluje, czy zostanie zadeklarowany błąd, jeżeli do argumentu ## zostanie przekazany ciąg niedający się zanalizować.

```
    public void testInvalidDouble() throws Exception {
      Args args = new Args("x##", new String[] {"-x","Czterdzieści dwa"});
      assertFalse(args.isValid());
      assertEquals(0, args.cardinality());
      assertFalse(args.has('x'));
      assertEquals(0, args.getInt('x'));
      assertEquals("Argument -x oczekuje wartości double, a było 'Czterdzieści dwa'.",
      args.errorMessage());
    }
  ---
```

```
  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Niedostępne.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Nie można znaleźć parametru znakowego dla -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c oczekuje liczby całkowitej, a był '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Nie można znaleźć parametru całkowitego dla -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c oczekuje liczby double, a był '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Nie można znaleźć parametru double dla -%c.",
                             errorArgumentId);
    }
    return "";
}
```

Wszystkie testy są wykonywane prawidłowo! Następny test sprawdza, czy prawidłowo wykrywamy brakujący argument `double`.

```
public void testMissingDouble() throws Exception {
  Args args = new Args("x##", new String[]{"-x"});
  assertFalse(args.isValid());
  assertEquals(0, args.cardinality());
  assertFalse(args.has('x'));
  assertEquals(0.0, args.getDouble('x'), 0.01);
  assertEquals("Nie można znaleźć parametru double dla -x.",
               args.errorMessage());
}
```

Ten również działa, jak oczekiwaliśmy. Napisaliśmy go tylko dla zachowania kompletności kodu.

Kod wyjątków jest dosyć brzydki i nie należy naprawdę do klasy `Args`. Zgłaszamy również `ParseException`, który na pewno nie należy do nas. Połączymy więc wszystkie wyjątki w jedną klasę `ArgsException` i przeniesiemy ją do osobnego modułu.

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
---
public class Args {
  ...
  private char errorArgumentId = '\0';
```

```
  private String errorParameter = "TILT";
  private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

private boolean parse() throws ArgsException {
  if (schema.length() == 0 && argsList.size() == 0)
    return true;
  parseSchema();
  try {
    parseArguments();
  } catch (ArgsException e) {
  }
  return valid;
}

private boolean parseSchema() throws ArgsException {
  ...
}

private void parseSchemaElement(String element) throws ArgsException {
  ...
  else
    throw new ArgsException(
      String.format("Argument: %c ma niewłaściwy format: %s.",
                    elementId,elementTail));
}

private void validateSchemaElementId(char elementId) throws ArgsException {
  if (!Character.isLetter(elementId)) {
    throw new ArgsException(
      "Zły znak:" + elementId + "w formacie Args: " + schema);
  }
}

...

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}

...

  private class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_STRING;
        throw new ArgsException();
```

```
      }
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }

  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
}
```

Teraz jest ładnie. Jedynym wyjątkiem zgłaszanym przez `Args` jest `ArgsException`. Przeniesienie `ArgsException` do osobnego modułu spowodowało, że możemy przenieść wiele dodatkowego kodu obsługi błędów do tego modułu, osobnego względem modułu `Args`. Zapewnia on naturalne i oczywiste miejsce na umieszczenie całego tego kodu i naprawdę pomaga wyczyścić moduł `Args`.

Dzięki temu wyjątki i kod obsługi błędów pozostają całkowicie oddzielone od modułu `Args` (patrz listingi 14.13 do 14.16). Zostało to osiągnięte w serii 30 małych kroków, pomiędzy którymi testy stale były wykonywane prawidłowo.

LISTING 14.13. ArgsTest.java

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
  public void testCreateWithNoSchemaOrArguments() throws Exception {
    Args args = new Args("", new String[0]);
    assertEquals(0, args.cardinality());
  }

  public void testWithNoSchemaButWithOneArgument() throws Exception {
    try {
      new Args("", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
      e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testWithNoSchemaButWithMultipleArguments() throws Exception {
    try {
      new Args("", new String[]{"-x", "-y"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
      e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testNonLetterSchema() throws Exception {
    try {
      new Args("*", new String[]{});
      fail("Konstruktor Args powinien zgłosić wyjątek");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
      e.getErrorCode());
      assertEquals('*', e.getErrorArgumentId());
    }
  }

  public void testInvalidArgumentFormat() throws Exception {
    try {
      new Args("f~", new String[]{});
      fail("Konstruktor Args powinien zgłosić wyjątek");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
      assertEquals('f', e.getErrorArgumentId());
    }
  }

  public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args("x", new String[]{"-x"});
    assertEquals(1, args.cardinality());
```

```
  assertEquals(true, args.getBoolean('x'));
}

public void testSimpleStringPresent() throws Exception {
  Args args = new Args("x*", new String[]{"-x", "param"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals("param", args.getString('x'));
}

public void testMissingStringArgument() throws Exception {
  try {
    new Args("x*", new String[]{"-x"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }
}

public void testSpacesInFormat() throws Exception {
  Args args = new Args("x, y", new String[]{"-xy"});
  assertEquals(2, args.cardinality());
  assertTrue(args.has('x'));
  assertTrue(args.has('y'));
}

public void testSimpleIntPresent() throws Exception {
  Args args = new Args("x#", new String[]{"-x", "42"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42, args.getInt('x'));
}

public void testInvalidInteger() throws Exception {
  try {
    new Args("x#", new String[]{"-x", "Czterdzieści dwa"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
    assertEquals("Czterdzieści dwa", e.getErrorParameter());
  }
}

public void testMissingInteger() throws Exception {
  try {
    new Args("x#", new String[]{"-x"});
    fail();
  } catch (ArgsException e) {
    assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
    assertEquals('x', e.getErrorArgumentId());
  }
}

public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[]{"-x", "42.3"});
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .OO1);
}

public void testInvalidDouble() throws Exception {
  try {
```

```
        new Args("x##", new String[]{"-x", "Czterdzieści dwa"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Czterdzieści dwa", e.getErrorParameter());
      }
    }

    public void testMissingDouble() throws Exception {
      try {
        new Args("x##", new String[]{"-x"});
        fail();
      } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
      }
    }
  }
```

LISTING 14.14. ArgsExceptionTest.java

```
  public class ArgsExceptionTest extends TestCase {
    public void testUnexpectedMessage() throws Exception {
      ArgsException e =
        new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                          'x', null);
      assertEquals("Argument -x nieoczekiwany.", e.errorMessage());
    }

    public void testMissingStringMessage() throws Exception {
      ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                          'x', null);
      assertEquals("Nie można znaleźć parametru znakowego dla -x.", e.errorMessage());
    }

    public void testInvalidIntegerMessage() throws Exception {
      ArgsException e =
      new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                        'x', "Czterdzieści dwa");
      assertEquals("Argument -x oczekuje liczby całkowitej 'Czterdzieści dwa'.",
      e.errorMessage());
    }

    public void testMissingIntegerMessage() throws Exception {
      ArgsException e =
        new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
        assertEquals("Nie można znaleźć parametru całkowitego dla -x.", e.errorMessage());
    }

    public void testInvalidDoubleMessage() throws Exception {
      ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                          'x', "Czterdzieści dwa");
      assertEquals("Argument -x oczekuje liczby double, a był 'Czterdzieści dwa'.",
      e.errorMessage());
    }

    public void testMissingDoubleMessage() throws Exception {
      ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                          'x', null);
      assertEquals("Nie można znaleźć parametru double dla -x.", e.errorMessage());
    }
  }
```

LISTING 14.15. ArgsException.java

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode, char errorArgumentId,
                       String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Niedostępne.");
      case UNEXPECTED_ARGUMENT:
        return String.format("Nieoczekiwany argument -%c.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Nie można znaleźć parametru znakowego dla -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c oczekuje liczby całkowitej, a był '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
```

```
          return String.format("Nie można znaleźć parametru całkowitego dla -%c.",
                               errorArgumentId);
        case INVALID_DOUBLE:
          return String.format("Argument -%c oczekuje liczby double, a był '%s'.",
                               errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
          return String.format("Nie można znaleźć parametru double dla -%c.",
                               errorArgumentId);
      }
      return "";
    }

    public enum ErrorCode {
      OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
      MISSING_STRING,
      MISSING_INTEGER, INVALID_INTEGER,
      MISSING_DOUBLE, INVALID_DOUBLE}
  }
```

LISTING 14.16. Args.java

```
  public class Args {
    private String schema;
    private Map<Character, ArgumentMarshaler> marshalers =
      new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private List<String> argsList;

    public Args(String schema, String[] args) throws ArgsException {
      this.schema = schema;
      argsList = Arrays.asList(args);
      parse();
    }

    private void parse() throws ArgsException {
      parseSchema();
      parseArguments();
    }

    private boolean parseSchema() throws ArgsException {
      for (String element : schema.split(",")) {
        if (element.length() > 0) {
          parseSchemaElement(element.trim());
        }
      }
      return true;
    }

    private void parseSchemaElement(String element) throws ArgsException {
      char elementId = element.charAt(0);
      String elementTail = element.substring(1);
      validateSchemaElementId(elementId);
      if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
      else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
      else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
      else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
      else
        throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                                elementId, elementTail);
```

```
}

private void validateSchemaElementId(char elementId) throws ArgsException {
  if (!Character.isLetter(elementId)) {
    throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                            elementId, null);
  }
}

private void parseArguments() throws ArgsException {
  for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
    String arg = currentArgument.next();
    parseArgument(arg);
  }
}

private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                            argChar, null);
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    e.setErrorArgumentId(argChar);
    throw e;
  }
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public boolean getBoolean(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
```

```
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }

  public String getString(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }

  public double getDouble(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }
}
```

Zmiany w klasie `Args` w większości polegały na usunięciu pewnych elementów. Sporo kodu przenieśliśmy z `Args` do `ArgsException`. Świetnie. Dodatkowo przenieśliśmy klasy `ArgumentMarshaler` do osobnych plików. Jeszcze lepiej!

Większość dobrych projektów programowych dotyczy partycjonowania — tworzenia odpowiednich miejsc, w których można umieścić różne części kodu. Takie rozdzielenie problemów powoduje, że kod jest znacznie prostszy do zrozumienia i utrzymania.

Szczególnie interesująca jest metoda `errorMessage` z `ArgsException`. Jasne było, że umieszczenie komunikatów błędów formatowania w `Args` było naruszeniem SRP. Klasa `Args` powinna dotyczyć przetwarzania argumentów, a nie formatowania komunikatów błędów. Jednak czy faktycznie ma sens umieszczanie formatowania komunikatów błędów w `ArgsException`?

W zasadzie jest to kompromis. Użytkownicy, którzy nie chcą korzystać z komunikatów błędów dostarczanych przez `ArgsException`, mogą napisać własne. Jednak wygoda posiadania gotowych komunikatów błędów nie jest bez znaczenia.

Teraz powinno być jasne, że jesteśmy bardzo blisko końcowego rozwiązania, przedstawionego na początku tego rozdziału. Ostatnie przekształcenia pozostawię do wykonania jako ćwiczenie.

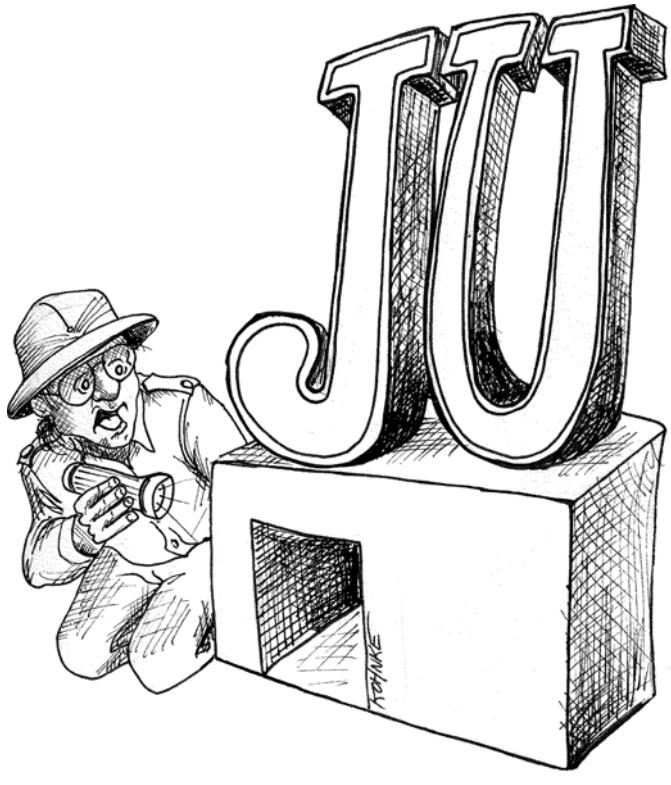
Zakończenie

Nie wystarczy napisać działający kod. Działający kod jest często fatalnie napisany. Programiści, którym wystarcza jedynie działający kod, działają nieprofesjonalnie. Mogą oni obawiać się, że nie będą mieli czasu na poprawienie struktury i projektu swojego kodu, ale nie mogą się z tym zgodzić. Nic nie ma tak głębokiego i w długim czasie degradującego wpływu na prowadzenie projektu, jak zły kod. Złe harmonogramy mogą być zmodyfikowane, złe wymagania przedefiniowane. Złą dynamikę zespołu da się poprawić. Jednak zły kod psuje się, stając się coraz większym ciężarem dla zespołu. Widziałem wiele zespołów, które się rozsypywały, ponieważ w pośpiechu tworzyły złośliwe bagnisko kodu, który na zawsze zdeterminował ich przeznaczenie.

Oczywiście, nieudany kod można wyczyścić. Jednak jest to bardzo kosztowne. Wraz z psuciem się kodu moduły przeplatają się ze sobą, tworząc wiele ukrytych i zagmatwanych zależności. Wyszukiwanie i przerywanie starych zależności jest czasochłonnym i żmudnym zadaniem. Z drugiej strony, zachowanie czystości kodu jest względnie łatwe. Jeżeli narobimy bałaganu w kodzie rano, łatwo możemy go posprzątać po południu. Jednak jeszcze łatwiej jest posprzątać od razu, gdy tylko nabałaganimy.

Tak więc rozwiązaniem jest ciągłe zachowywanie czystości i maksymalnej prostoty kodu. Nigdy nie pozwalajmy, by zaczął się psuć.

Struktura biblioteki JUnit



JUnit jest jedną z najsłynniejszych bibliotek języka Java. Jest on oparty na prostej koncepcji, precyzyjnej definicji i eleganckiej implementacji. Jak jednak wygląda jego kod? W tym rozdziale poddamy krytyce przykładowy kod biblioteki JUnit.

Biblioteka JUnit

JUnit ma wielu autorów, ale powstał na podstawie pracy Kent Becka i Erica Gamma w czasie ich wspólnego lotu do Atlanty. Kent chciał się nauczyć języka Java, a Eric chciał poznać bibliotekę testów dla języka Smalltalk, której autorem był Kent. „Co mogło być bardziej naturalnego dla maniaków uwięzionych w fotelach, jak tylko wyciągnąć swoje laptopy i zacząć kodować?”[1]. Po trzech godzinach pracy na wysokim pułapie napisali oni podstawy JUnit.

Moduł, którym się zajmiemy, jest sprytnym kawałkiem kodu pomagającym identyfikować błędy porównania napisów. Nosi on nazwę `ComparisonCompactor`. Analizując dwa różniące się napisy, takie jak `ABCDE` i `ABXDE`, wskazuje on różnice przez wygenerowanie ciągu takiego jak `<...B[X]D...>`.

Omówimy to dokładniej nieco dalej, a najlepszym wyjaśnieniem będą przypadki testowe. Spójrzmy więc na kod z listingu 15.1, który szczegółowo wyjaśnia wymagania dotyczące tego modułu. Możemy ocenić strukturę testów. Czy nie mogły być one prostsze lub bardziej oczywiste?

LISTING 15.1. ComparisonCompactorTest.java

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

  public void testMessage() {
    String failure= new ComparisonCompactor(0, "b", "c").compact("a");
    assertTrue("a oczekiwane:<[b]> a było:<[c]>".equals(failure));
  }

  public void testStartSame() {
    String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
    assertEquals("oczekiwane:<b[a]> a było:<b[c]>", failure);
  }

  public void testEndSame() {
    String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
    assertEquals("oczekiwane:<[a]b> a było:<[c]b>", failure);
  }

  public void testSame() {
    String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
    assertEquals("oczekiwane:<ab> a było:<ab>", failure);
  }

  public void testNoContextStartAndEndSame() {
    String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
    assertEquals("oczekiwane:<...[b]...> a było:<...[d]...>", failure);
  }

  public void testStartAndEndContext() {
    String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
    assertEquals("oczekiwane:<a[b]c> a było:<a[d]c>", failure);
  }
  public void testStartAndEndContextWithEllipses() {
    String failure=
```

[1] Kent Beck, *JUnit Pocket Guide*, O'Reilly 2004, s. 43.

```
        new ComparisonCompactor(1, "abcde", "abfde").compact(null);
      assertEquals("oczekiwane:<...b[c]d...> a było:<...b[f]d...>", failure);
    }

    public void testComparisonErrorStartSameComplete() {
      String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
      assertEquals("oczekiwane:<ab[]> a było:<ab[c]>", failure);
    }

    public void testComparisonErrorEndSameComplete() {
      String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
      assertEquals("oczekiwane:<[]...> a było:<[a]...>", failure);
    }

    public void testComparisonErrorEndSameCompleteContext() {
      String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
      assertEquals("oczekiwane:<[]bc> a było:<[a]bc>", failure);
    }

    public void testComparisonErrorOverlapingMatches() {
      String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
      assertEquals("oczekiwane:<...[]...> a było:<...[b]...>", failure);
    }

    public void testComparisonErrorOverlapingMatchesContext() {
      String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
      assertEquals("oczekiwane:<ab[]c> a było:<ab[b]c>", failure);
    }

    public void testComparisonErrorOverlapingMatches2() {
      String failure= new ComparisonCompactor(0, "abcdde", "abcde").compact(null);
      assertEquals("oczekiwane:<...[d]...> a było:<...[]...>", failure);
    }

    public void testComparisonErrorOverlapingMatches2Context() {
      String failure=
        new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
      assertEquals("oczekiwane:<...cd[d]e> a było:<...cd[]e>", failure);
    }

    public void testComparisonErrorWithActualNull() {
      String failure= new ComparisonCompactor(0, "a", null).compact(null);
      assertEquals("oczekiwane:<a> a było:<null>", failure);
    }

    public void testComparisonErrorWithActualNullContext() {
      String failure= new ComparisonCompactor(2, "a", null).compact(null);
      assertEquals("oczekiwane:<a> a było:<null>", failure);
    }

    public void testComparisonErrorWithExpectedNull() {
      String failure= new ComparisonCompactor(0, null, "a").compact(null);
      assertEquals("oczekiwane:<null> a było:<a>", failure);
    }

    public void testComparisonErrorWithExpectedNullContext() {
      String failure= new ComparisonCompactor(2, null, "a").compact(null);
      assertEquals("oczekiwane:<null> a było:<a>", failure);
    }

    public void testBug609972() {
      String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
      assertEquals("oczekiwane:<[S&P50]0> a było:<[]0>", failure);
    }
}
```

Uruchomiłem analizę pokrycia kodu `ComparisionCompactor` przy użyciu tych testów. Kod jest w stu procentach pokryty. Każdy wiersz kodu, każda instrukcja `if` i pętla `for` jest wykonywana w tych testach. Daje mi to duży stopień pewności, że kod działa, i wzbudza we mnie duży respekt dla umiejętności autorów.

Kod `ComparisionCompactor` jest zamieszczony na listingu 15.2. Warto poświęcić trochę czasu na zapoznanie się z tym kodem. Chyba wszyscy zauważą, że jest ładnie podzielony, wyrazisty i ma prostą strukturę. Po zakończeniu tej analizy zbierzemy wszystkie uwagi.

LISTING 15.2. ComparisonCompactor.java (Oryginalny)

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int fContextLength;
  private String fExpected;
  private String fActual;
  private int fPrefix;
  private int fSuffix;

  public ComparisonCompactor(int contextLength,
                             String expected,
                             String actual) {
    fContextLength = contextLength;
    fExpected = expected;
    fActual = actual;
  }

  public String compact(String message) {
    if (fExpected == null || fActual == null || areStringsEqual())
      return Assert.format(message, fExpected, fActual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(fExpected);
    String actual = compactString(fActual);
    return Assert.format(message, expected, actual);
  }

  private String compactString(String source) {
    String result = DELTA_START +
                    source.substring(fPrefix, source.length() -
                      fSuffix + 1) + DELTA_END;
    if (fPrefix > 0)
      result = computeCommonPrefix() + result;
    if (fSuffix > 0)
      result = result + computeCommonSuffix();
    return result;
  }

  private void findCommonPrefix() {
    fPrefix = 0;
    int end = Math.min(fExpected.length(), fActual.length());
    for (; fPrefix < end; fPrefix++) {
      if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
```

```
        break;
      }
    }

    private void findCommonSuffix() {
      int expectedSuffix = fExpected.length() - 1;
      int actualSuffix = fActual.length() - 1;
      for (;
           actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
           actualSuffix--, expectedSuffix--) {
        if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
          break;
      }
      fSuffix = fExpected.length() - expectedSuffix;
    }

    private String computeCommonPrefix() {
      return (fPrefix > fContextLength ? ELLIPSIS : "") +
               fExpected.substring(Math.max(0, fPrefix - fContextLength),
                                   fPrefix);
    }

    private String computeCommonSuffix() {
      int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
      fExpected.length());
      return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
             (fExpected.length() - fSuffix + 1 < fExpected.length() -
              fContextLength ? ELLIPSIS : "");
    }

    private boolean areStringsEqual() {
      return fExpected.equals(fActual);
    }
  }
```

Można mieć pewne zastrzeżenia do tego modułu. Znajduje się tu nieco długich wyrażeń, kilka dziwnych operacji +1 i tak dalej. Jednak ogólnie jest to całkiem niezły moduł. Ostatecznie mógłby on wyglądać jak na listingu 15.3.

LISTING 15.3. ComparisonCompator.java (uszkodzony)

```
package junit.framework;

public class ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;

  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
      return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
```

```
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
        break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
  }

  private String compactString(String s) {
    String result =
      "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
      result = (pfx > ctxt ? "..." : "") +
        s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
    if (sfx > 0) {
      int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
      result = result + (s1.substring(s1.length() - sfx + 1, end) +
        (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
    }
    return result;
  }
}
```

Mimo że autorzy nadali temu modułowi bardzo dobry kształt, *zasada skautów*[2] mówi, że powinniśmy pozostawić go czystszym, niż go zastaliśmy. W jaki sposób można poprawić oryginalny kod z listingu 15.2?

Pierwszą rzeczą, jakiej nie cierpię, jest prefiks `f` przed zmiennymi składowymi [N6]. Dzisiejsze środowiska powodują, że taki sposób kodowania zakresu jest nadmiarowy. Wyeliminujemy więc wszystkie `f`.

```
private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;
```

Następnie mamy niehermetyzowany warunek na początku funkcji `compact` [G28].

```
public String compact(String message) {
  if (expected == null || actual == null || areStringsEqual())
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}
```

[2] Patrz „Zasada skautów” w rozdziale 1.

Warunek ten powinien być hermetyzowany, aby nasze intencje były jasne. Z tego powodu wyodrębnimy metodę, która je wyjaśnia.

```
public String compact(String message) {
  if (shouldNotCompact())
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {
  return expected == null || actual == null || areStringsEqual();
}
```

Nie podoba mi się notacja `this.expected` oraz `this.actual` w funkcji `compact`. Powstała ona, gdy zmieniłem nazwę z `fExpected` na `expected`. Dlaczego w tej funkcji wykorzystane są nazwy mające takie same nazwy co zmienne składowe? Czy nie reprezentują czegoś innego [N4]? Powinniśmy użyć bardziej znaczących nazw.

```
String compactExpected = compactString(expected);
String compactActual = compactString(actual);
```

Wyrażenia negatywne są nieco trudniejsze do zrozumienia niż pozytywne [G29]. Postawimy więc wyrażenie `if` na głowie i zmienimy sens warunku.

```
public String compact(String message) {
  if (canBeCompacted()) {
    findCommonPrefix();
    findCommonSuffix();
    String compactExpected = compactString(expected);
    String compactActual = compactString(actual);
    return Assert.format(message, compactExpected, compactActual);
  } else {
    return Assert.format(message, expected, actual);
  }
}

private boolean canBeCompacted() {
  return expected != null && actual != null && !areStringsEqual();
}
```

Nazwa funkcji jest dość dziwna [N7]. Choć wykonuje ona zmniejszanie ciągów, w rzeczywistości może tego nie robić, jeżeli `canBeCompacted` zwróci `false`. Tak więc nazwanie tej funkcji `compact` powoduje ukrycie efektu ubocznego kontroli błędów. Należy również zauważyć, że funkcja zwraca sformatowany komunikat, a nie po prostu zmniejszony ciąg. Dlatego nazwa tej funkcji powinna brzmieć `formatCompactedComparison`. Będzie znacznie czytelniejsza, gdy złączymy ją z argumentem funkcji:

```
public String formatCompactedComparison(String message) {
```

Faktyczne zmniejszanie oczekiwanego i otrzymanego ciągu znajduje się w ciele instrukcji `if`. Powinniśmy wyodrębnić tę operację do osobnej metody o nazwie `compactExpectedAndActual`. Jednak chcemy, aby funkcja `formatCompactedComparison` wykonywała całe formatowanie. Funkcja `compact...` nie powinna robić nic poza zmniejszaniem [G30]. Podzielmy kod w następujący sposób:

```
...
  private String compactExpected;
  private String compactActual;
...
  public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
      compactExpectedAndActual();
      return Assert.format(message, compactExpected, compactActual);
    } else {
      return Assert.format(message, expected, actual);
    }
  }

  private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
```

Trzeba zauważyć, że wymaga to wypromowania `compactExpected` oraz `compactActual` jako zmiennych składowych. Nie lubię sposobu, w jaki w ostatnich dwóch wierszach nowej funkcji są zwracane zmienne, a w dwóch pierwszych nie. Nie jest tu używana spójna konwencja [G11]. Powinniśmy zmienić `findCommonPrefix` oraz `findCommonSuffix`, aby zwracały wartość przedrostka i przyrostka.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonPrefix() {
  int prefixIndex = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex < end; prefixIndex++) {
    if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
      break;
  }
  return prefixIndex;
}

private int findCommonSuffix() {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

Powinniśmy również zmienić nazwy zmiennych składowych na bardziej trafne [N1]; w końcu obie są indeksami.

Dokładna analiza pokazuje, że `findCommonSuffix` ujawnia *ukryte sprzężenie czasowe* [G31]; zmienna zależy od wyliczenia wartości `prefixIndex` przez `findCommonPrefix`. Jeżeli te dwie funkcje zostałyby wywołane w niewłaściwej kolejności, można by się spodziewać trudnej sesji debugowania.

Tak więc, aby ujawnić to sprzężenie czasowe, funkcja `findCommonSuffix` będzie oczekiwała jako argumentu `prefixIndex`.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix(prefixIndex);
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonSuffix(int prefixIndex) {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

Nie jestem zadowolony z tego kodu. Przekazywanie argumentu `prefixIndex` jest zbyt dowolne [G32]. Pozwala określić kolejność, ale nie wyjaśnia potrzeby takiego uporządkowania. Inny programista może wycofać wprowadzone przez nas zmiany, ponieważ nie ma żadnej wskazówki, że parametr jest naprawdę potrzebny. Przyjmijmy więc inne podejście.

```
private void compactExpectedAndActual() {
  findCommonPrefixAndSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (;
       actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--
      ) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  suffixIndex = expected.length() - expectedSuffix;
}

private void findCommonPrefix() {
  prefixIndex = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex < end; prefixIndex++)
    if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
      break;
}
```

Umieściliśmy `findCommonPrefix` oraz `findCommonSuffix` z powrotem tam, gdzie były wcześniej, zmieniając nazwę `findCommonSuffix` na `findCommon`**`PrefixAnd`**`Suffix` i wywołując `findCommonPrefix` przed wykonaniem jakichkolwiek innych operacji. Uwypukla to czasową naturę sprzężenia tych dwóch funkcji w bardziej dramatyczny sposób niż w poprzednim rozwiązaniu. Wskazuje również, jak nieudana jest funkcja `findCommonPrefixAndSuffix`. Zajmijmy się jej wyczyszczeniem.

```
private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  int suffixLength = 1;
  for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
      break;
  }
  suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
  return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
  return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
}
```

Teraz jest znacznie lepiej. Ujawnione zostało, że `suffixIndex` jest w rzeczywistości długością przyrostka, czyli nie jest dobrze nazwana. To samo dotyczy `prefixIndex`, czyli w tym przypadku `index` i `length` są synonimami. Kod zyska na spójności, gdy będziemy korzystać z `length`. Jednak problem polega na tym, że `suffixIndex` nie zaczyna się od zera — zaczyna się od 1, czyli nie jest to prawdziwa długość. Przyczyna się to również do tego, że w `computeCommonSuffix` są użyte te wszystkie `+1` [G33]. Poprawmy więc to. Wynik jest zamieszczony na listingu 15.4.

LISTING 15.4. ComparisonCompactor.java (tymczasowy)

```
public class ComparisonCompactor {
...
  private int suffixLength;
...
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength <= prefixLength ||
    expected.length() - suffixLength <= prefixLength;
  }

...
  private String compactString(String source) {
    String result =
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END;
```

```
    if (prefixLength > 0)
      result = computeCommonPrefix() + result;
    if (suffixLength > 0)
      result = result + computeCommonSuffix();
    return result;
  }

...
  private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength +
    contextLength, expected.length()
    );
    return
      expected.substring(expected.length() - suffixLength, end) +
      (expected.length() - suffixLength <
        expected.length() - contextLength ?
        ELLIPSIS : "");
  }
```

Zamieniliśmy `+1` w `computeCommonSuffix` na `-1` w `charFromEnd`, gdzie jest to całkiem sensowne, oraz dwa operatory `<=` w `suffixOverlapsPrefix`, gdzie również jest to sensowne. Pozwoliło to nam zmienić nazwę `suffixIndex` na `suffixLength`, co znacznie poprawiło czytelność kodu.

Istnieje tu jednak problem. Po wyeliminowaniu `+1` pozostał w `compactString` następujący wiersz:

```
if (suffixLength > 0)
```

Spójrzmy na listing 15.4. Ponieważ `suffixLength` ma teraz wartość o jeden mniejszą niż wcześniej, powinniśmy zmienić operator `>` na `>=`. Jednak nie miało to sensu. *Teraz* ma! Jeżeli wcześniej nie miało to sensu, był to prawdopodobnie błąd. Cóż, właściwie nie był to błąd. Dokładna analiza pokazuje, że instrukcja `if` zapobiega teraz dołączaniu przyrostków o zerowej długości. Przed wprowadzeniem przez nas zmian instrukcja `if` nie działała, ponieważ zmienna `suffixIndex` nigdy nie przyjmowała wartości mniejszej od jeden!

Dotyczy to *obu* instrukcji `if` w `compactString`! Wygląda na to, że można je wyeliminować. Zakomentujemy je więc i uruchomimy testy. Zostały wykonane prawidłowo! Możemy więc zmodyfikować `compactString`, eliminując nadmiarowe instrukcje `if`, co znacznie upraszcza tę funkcję [G9].

```
private String compactString(String source) {
  return
    computeCommonPrefix() +
    DELTA_START +
    source.substring(prefixLength, source.length() - suffixLength) +
    DELTA_END +
    computeCommonSuffix();
}
```

Teraz jest znacznie lepiej! Widzimy, że funkcja `compactString` po prostu łączy ze sobą fragmenty tekstu. Z pewnością możemy pokazać to jeszcze jaśniej, wykonać jeszcze wiele małych usprawnień. Jednak zamiast pokazywać pozostałe zmiany, zamieścimy ostateczny wynik na listingu 15.5.

LISTING 15.5. ComparisonCompactor.java (ostateczny)

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int contextLength;
  private String expected;
  private String actual;
  private int prefixLength;
  private int suffixLength;

  public ComparisonCompactor(
    int contextLength, String expected, String actual
  ) {
    this.contextLength = contextLength;
    this.expected = expected;
    this.actual = actual;
  }

  public String formatCompactedComparison(String message) {
    String compactExpected = expected;
    String compactActual = actual;
    if (shouldBeCompacted()) {
      findCommonPrefixAndSuffix();
      compactExpected = compact(expected);
      compactActual = compact(actual);
    }
    return Assert.format(message, compactExpected, compactActual);
  }

  private boolean shouldBeCompacted() {
    return !shouldNotBeCompacted();
  }

  private boolean shouldNotBeCompacted() {
    return expected == null ||
           actual == null ||
           expected.equals(actual);
  }

  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength)
      )
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix() {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }
```

Więcej na: www.ebook4all.pl

```
  private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
      if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
        break;
  }

  private String compact(String s) {
    return new StringBuilder()
      .append(startingEllipsis())
      .append(startingContext())
      .append(DELTA_START)
      .append(delta(s))
      .append(DELTA_END)
      .append(endingContext())
      .append(endingEllipsis())
      .toString();
  }

  private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
  }

  private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
  }

  private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
  }

  private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
      Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
  }

  private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
  }
}
```

Kod ten jest całkiem ładny. Moduł ma oddzieloną grupę funkcji analitycznych i inną grupę funkcji syntetycznych. Są one posortowane topologicznie, dzięki czemu definicja każdej z funkcji znajduje się zaraz po jej użyciu. Wszystkie funkcje analityczne znajdują się na początku, a funkcje syntetyczne na końcu.

Dokładniejsza analiza kodu pokazuje, że cofnąłem kilka decyzji, jakie podjąłem we wcześniejszej części tego rozdziału. Na przykład wbudowałem kilka wyodrębnionych metod z powrotem do `formatCompactedComparison` i zmieniłem sens wyrażenia `shouldNotBeCompacted`. Jest to typowe. Często jedna przebudowa prowadzi do anulowania wcześniejszej. Przebudowa jest procesem iteracyjnym pełnym prób i błędów — procesem, który wynika z naszego dążenia do profesjonalizmu.

Zakończenie

Wypełniliśmy regułę skauta. Pozostawiliśmy moduł czyściejszy, niż go dostaliśmy. Nie można powiedzieć, że wcześniej nie był czysty. Jego autorzy świetnie się spisali. Jednak nie ma modułu, którego nie można poprawić, a każdy z nas jest odpowiedzialny za pozostawienie kodu w lepszym stanie, niż był wcześniej.

Przebudowa klasy SerialDate

Jeżeli otworzymy stronę *http://www.jfree.org/jcommon/index.php*, znajdziemy tam bibliotekę JCommon. Głęboko w tej bibliotece znajdziemy pakiet o nazwie `org.jfree.date`. Z kolei w pakiecie znajduje się klasa o nazwie `SerialDate`. W niniejszym rozdziale przeanalizujemy tę klasę.

Autorem klasy `SerialDate` jest David Gilbert. Jasne jest, że David to doświadczony i kompetentny programista. Jak możemy zauważyć, podczas tworzenia tego kodu wykazał się dyscypliną i profesjonalizmem. W przypadku wszystkich zastosowań jest to „dobry kod". A ja zamierzam rozebrać go na części.

Nie jest to z mojej strony żadna złośliwość. Nie uważam również, że jestem znacznie lepszym programistą niż David, więc mam prawo przeprowadzić sąd nad jego kodem. Gdyby ktoś przeanalizował mój kod, jestem pewny, że znalazłby wiele fragmentów, do których miałby uwagi.

Nie, nie jest to akt złośliwości ani arogancji. Moim zamiarem jest tylko profesjonalna recenzja. Jest to coś, co każdy z nas powinien wykonywać bez problemów. Jest to również coś, z czego powinniśmy być zadowoleni, gdy ktoś to robi dla nas. Tylko przez taką krytykę możemy się czegoś nauczyć. Lekarze to robią. Piloci to robią. Prawnicy to robią. Również my, programiści, powinniśmy się nauczyć recenzować dzieła innych.

Jeszcze jedno słowo na temat Davida Gilberta. David to więcej niż tylko dobry programista. Miał odwagę i dobrą wolę, aby za darmo zaoferować swój kod społeczności. Udostępnił go dla wszystkich, umożliwiając jego użycie i publiczne badanie. Jest to świetnie zrobiony kod!

Klasa `SerialDate` (listing B.1 w dodatku B) reprezentuje datę w języku Java. Po co nam kolejna klasa reprezentująca datę, skoro Java posiada już między innymi klasy `java.util.Date` oraz `java.util.Calendar`? Autor napisał tę klasę w odpowiedzi na problemy, z jakimi często się spotykaliśmy. Otwierający komentarz Javadoc (wiersz 67.) jasno to wyjaśnia. Możemy mieć wątpliwości na temat jego intencji, ale ja kiedyś musiałem zmagać się z tym problemem i z radością powitałem klasę, która operuje na datach zamiast na czasie.

Na początek uruchamiamy

W klasie `SerialDateTests` (listing B.2 w dodatku B) znajduje się kilka testów jednostkowych. Wszystkie te testy są wykonywane prawidłowo. Niestety, szybki przegląd testów pokazuje, że nie testują wszystkiego [T1]. Na przykład wyszukanie zastosowań dla metody `MonthCodeToQuarter` (wiersz 334.) wykazuje, że nie jest używana [F4]. Zatem testy jednostkowe nie testują jej.

Skorzystałem zatem z programu Clover, aby sprawdzić, co jest pokryte przez testy jednostkowe, a co nie. Clover wykazał, że testy jednostkowe wykonują tylko 91 ze 185 instrukcji wykonywalnych w klasie `SerialDate` (~50 procent) [T2]. Mapa pokrycia wyglądała jak dziurawy koc, z wielkimi połaciami niewykonywanego kodu rozsianymi po całej klasie.

Musiałem dokładnie zrozumieć, jak przebudować tę klasę. Nie mogłem tego robić bez znacznie większego pokrycia kodu testami. Napisałem więc własny zestaw zupełnie niezależnych testów jednostkowych (listing B.4 w dodatku B).

Przeglądając te testy, można zauważyć, że wiele z nich jest zakomentowanych. Testy te nie wykonują się prawidłowo. Reprezentują one te funkcje, które według mnie powinna posiadać klasa `SerialDate`. Dlatego w czasie przebudowy `SerialDate` będziemy dążyć do uruchomienia również tych testów.

Nawet pomimo zakomentowania części testów Clover raportuje, że nowe testy jednostkowe wykonują 170 ze 180 (92 procent) instrukcji wykonywalnych. Jest to całkiem niezły wynik i uważam, że jestem w stanie uzyskać wyższy.

Kilka pierwszych zakomentowanych testów (wiersze od 23. do 63.) było z mojej strony nieco przesadzonych. Program nie był zaprojektowany, aby przeszedł te testy, ale działanie takie wydawało mi się oczywiste [G2]. Nie jestem pewien, dlaczego metoda `testWeekdayCodeToString` była napisana

jako pierwsza, ale ponieważ już jest, wydaje się oczywiste, że powinna rozpoznawać wielkość liter. Napisanie tych testów było bardzo łatwe [T3]. Spowodowanie, aby się wykonywały, było równie proste — po prostu zmieniłem wiersze 259. i 263., aby wykorzystywana była metoda `equalsIgnoreCase`.

Pozostawiłem zakomentowane testy w wierszach 32. i 45., ponieważ nie było dla mnie jasne, czy skróty „tues" oraz „thurs" powinny być obsługiwane.

Testy z wierszy 153. i 154. nie były wykonywane prawidłowo. Jasne jest, że powinny [G2]. Możemy łatwo to poprawić, razem z testami z wierszy od 163. do 213., przez wprowadzenie następujących zmian do funkcji `stringToMonthCode`:

```
457       if ((result < 1) || (result > 12)) {
            result = -1;
458         for (int i = 0; i < monthNames.length; i++) {
459           if (s.equalsIgnoreCase(shortMonthNames[i])) {
460               result = i + 1;
461             break;
462           }
463           if (s.equalsIgnoreCase(monthNames[i])) {
464             result = i + 1;
465             break;
466           }
467         }
468       }
```

Zakomentowany test z wiersza 318. ujawnia błąd w metodzie `getFollowingDayOfWeek` (wiersz 672.). 25 grudnia 2004 był sobotą. Następną sobotą był 1 stycznia 2005. Jednak po uruchomieniu testów okazuje się, że `getFollowingDayOfWeek` zwraca 25 grudnia jako sobotę następującą po 25 grudnia. Oczywiście jest to błąd [G3], [T1]. Problem znajduje się w wierszu 685. Jest to typowy błąd warunku granicznego [T1]. Powinien on wyglądać następująco:

```
685       if (baseDOW >= targetWeekday) {
```

Warto zauważyć, że funkcja ta była celem wcześniejszej poprawki. Historia zmian wykazuje (wiersz 43.), że te „błędy" były poprawione w `getPreviousDayOfWeek`, `getFollowingDayOfWeek` oraz `getNearestDayOfWeek` [T6].

Test jednostkowy `testGetNearestDayOfWeek` (wiersz 329.), który testuje metodę `getNearest`
`↪DayOfWeek` (wiersz 705.), nie był tak długi i dokładny, jak jest obecnie. Dodałem do niego wiele przypadków testowych, ponieważ nie wszystkie moje początkowe przypadki testowe wykonywały się prawidłowo [T6]. Można zauważyć wzorzec awarii przez sprawdzenie, które przypadki testowe są zakomentowane. Wzorzec ten wiele nam mówi [T7]. Pokazuje, że algorytm nie działa prawidłowo, jeżeli najbliższy dzień jest w przyszłości. Jasne jest, że jest to jakiś rodzaj błędu w warunku granicznym [T5].

Równie interesujący jest wzór pokrycia testami raportowany przez program Clover [T8]. Wiersz 719. nigdy nie jest wykonywany! Oznacza to, że instrukcja `if` w wierszu 718. zawsze ma wartość `false`. Patrząc na kod, można stwierdzić, że musi mieć ona wartość `true`. Zmienna `adjust` jest zawsze ujemna, więc nie może być większa lub równa 4. A zatem algorytm ten jest nieprawidłowy.

Prawidłowy algorytm jest przedstawiony poniżej:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
  adjust -= 7;

return SerialDate.addDays(adjust, base);
```

Testy z wierszy 417. oraz 429. mogą być wykonywane prawidłowo przez zgłoszenie wyjątku `IllegalArgumentException`, zamiast zwracać komunikat błędu z `weekInMonthToString` oraz `relativeToString`.

Po wprowadzeniu tych zmian wszystkie testy jednostkowe są wykonywane prawidłowo i uważam, że `SerialDate` teraz działa. Czas więc, by była napisana „właściwie".

Teraz poprawiamy

Mam zamiar przejść od początku do końca klasy `SerialDate`, ulepszając kolejne konstrukcje. Choć nie jest to pokazane, po każdej wprowadzonej zmianie uruchamiałem wszystkie testy jednostkowe `JCommon`, razem z moimi ulepszonymi testami jednostkowymi `SerialDate`. Pozostałe testy upewniały mnie, że każda pokazana tu zmiana działa z pozostałym kodem `JCommon`.

Zaczynając od wiersza 1., widzimy blok komentarzy z informacjami o licencji, autorach i historii zmian. Zgadzam się, że określone problemy prawne powinny zostać rozwiązane, więc informacje o prawach autorskich i licencjach powinny zostać. Z drugiej strony, historia zmian jest pozostałością z lat sześćdziesiątych ubiegłego stulecia. Mamy teraz narzędzia kontroli wersji, które robią to za nas. Historia ta powinna zostać usunięta [C1].

Lista importów zaczynająca się od wiersza 61. powinna zostać skrócona przez użycie `java.text.*` oraz `java.util.*` [J1].

Formatowanie HTML w Javadoc przyprawia mnie o ból głowy (wiersz 67.). Pliki źródłowe z więcej niż jednym językiem są dla mnie problemem. Ten komentarz korzysta z *czterech* języków: Java, angielskiego, Javadoc oraz HTML [G1]. Przy tak dużej liczbie zastosowanych języków trudno zachować prostotę. Na przykład ładne pozycjonowanie w wierszach 71. i 72. jest tracone po wygenerowaniu Javadoc, a poza tym kto chce widzieć w kodzie źródłowym znaczniki `<ul>` i `<li>`. Lepszą strategią jest objęcie całego komentarza znacznikami `<pre>`, aby formatowanie naturalne dla kodu źródłowego zostało zachowane w Javadoc[1].

W wierszu 86. znajduje się deklaracja klasy. Dlaczego klasa ta jest nazwana `SerialDate`? Jakie jest znaczenie wartości słowa „serial"? Czy pochodzi ono od `Serializable`? Raczej mało prawdopodobne.

---

[1] Jeszcze lepszym rozwiązaniem byłoby, gdyby Javadoc prezentował wszystkie komentarze w postaci wstępnie sformatowanej — te same komentarze wyglądały identycznie w kodzie i dokumentacji.

Nie chcę trzymać dalej Czytelników w niepewności. Wiem (lub co najmniej uważam, że wiem), dlaczego zostało użyte słowo „serial". Powodem jest użycie stałych `SERIAL_LOWER_BOUND` i `SERIAL_UPPER_BOUND` z wierszy 98. i 101. Jeszcze lepszą wskazówką jest komentarz zaczynający się w wierszu 830. Klasa ta jest nazwana `SerialDate`, ponieważ jest zaimplementowana z użyciem „numeru seryjnego", który jest liczbą dni od 30 grudnia 1899.

Są z tym związane dwa problemy. Po pierwsze, nazwa „numer seryjny" nie jest tak naprawdę prawidłowa. Może to być wątpliwe, ponieważ reprezentacja ta jest bardziej względnym przesunięciem niż numerem seryjnym. Nazwa „numer seryjny" ma więcej wspólnego ze znacznikami identyfikującymi produkty niż z datami. Dlatego nie uważam, aby nazwa ta była odpowiednio opisowa [N1]. Właściwszym terminem może być „ordinal" (kolejność).

Drugi problem jest ważniejszy. Nazwa `SerialDate` zakłada implementację. Klasa ta jest klasą abstrakcyjną. Nie ma potrzeby, aby zakładała cokolwiek na temat jej implementacji. W rzeczywistości istnieją powody, aby ukryć implementację! Uważam więc, że nazwa ta znajduje się na niewłaściwym poziomie abstrakcji [N2]. Według mnie klasa ta powinna nazywać się po prostu `Date`.

Niestety, w bibliotece Java znajduje się zbyt wiele klas nazwanych `Date`, więc nie jest to najlepsza nazwa. Ponieważ klasa ta operuje na dniach, a nie na czasie, rozważałem nazwanie jej `Day`, ale ta nazwa jest równie intensywnie używana w innych miejscach. W końcu wybrałem `DayDate` jako najlepszy kompromis.

Od teraz w rozdziale będę korzystał z nazwy `DayDate`. Czytelnik winien pamiętać, że listingi, na jakie będzie patrzył, nadal korzystają z nazwy `SerialDate`.

Rozumiem, dlaczego `DayDate` dziedziczy po `Comparable` i `Serializable`. Dlaczego jednak dziedziczy po `MonthConstants`? Klasa `MonthConstants` (listing B.3 w dodatku B) jest po prostu zbiorem statycznych stałych `final`, które definiują nazwy miesięcy. Dziedziczenie po klasach ze stałymi jest starą sztuczką używaną przez programistów Java, aby uniknąć używania wyrażeń takich jak `MonthConstants.January`, ale jest to zły pomysł [J2]. `MonthConstants` powinien być typem wyliczeniowym.

```
public abstract class DayDate implements Comparable,
                                         Serializable {
  public static enum Month {
    JANUARY(1),
    FEBRUARY(2),
    MARCH(3),
    APRIL(4),
    MAY(5),
    JUNE(6),
    JULY(7),
    AUGUST(8),
    SEPTEMBER(9),
    OCTOBER(10),
    NOVEMBER(11),
    DECEMBER(12);

    Month(int index) {
      this.index = index;
    }
```

```
    public static Month make(int monthIndex) {
      for (Month m : Month.values()) {
        if (m.index == monthIndex)
          return m;
      }
      throw new IllegalArgumentException("Niewłaściwy indeks miesiąca " + monthIndex);
    }
    public final int index;
  }
```

Zmiana `MonthConstants` na `enum` wymusza niewiele zmian w klasie `DayDate` i klasach jej używających. Wprowadzenie wszystkich tych zmian zajęło mi godzinę. Jednak każda funkcja, która wcześniej oczekiwała `int` z miesiącem, obecnie oczekuje enumeratora `Month`. Dzięki temu można usunąć metodę `isValidMonthCode` (wiersz 326.) i wszystkie sprawdzenia kodu miesiąca, takie jak te używane w `monthCodeToQuarter` (wiersz 356.) [G5].

Następnie w wierszu 91. mamy `serialVersionUID`. Zmienna ta jest używana do sterowania serializacją. Jeżeli ją zmienimy, to dowolny obiekt `DayDate` zapisany w starszej wersji oprogramowania nie będzie czytelny w nowszej i spowoduje zgłoszenie wyjątku `InvalidClassException`. Jeżeli nie zadeklarujemy zmiennej `serialVersionUID`, kompilator automatycznie ją dla nas wygeneruje i będzie ona miała inną wartość po każdej zmianie w module. Wiem, że we wszystkich dokumentach zalecana jest ręczna kontrola nad tą zmienną, ale wydaje mi się, że automatyczne sterowanie serializacją jest znacznie bezpieczniejsze [G4]. W końcu raczej będę debugował `InvalidClass` ↪`Exception`, niż podejmę dziwne działanie, jakie będzie wynikiem braku zmiany wartości `serial` ↪`VersionUID`. Dlatego usunę tę zmienną — przynajmniej na jakiś czas[2].

Uważam, że komentarz z wiersza 93. jest nadmiarowy. Nadmiarowe komentarze są po prostu dezinformującymi miejscami gromadzącymi kłamstwa [C2]. Zamierzam więc usunąć go.

Komentarze z wierszy 97. i 100. traktują o numerach seryjnych, o czym wspominałem wcześniej [C1]. Zmienne, jakie opisują, są najwcześniejszą i najpóźniejszą możliwą datą, jaką można opisać za pomocą `DayDate`. Może to być nieco jaśniejsze [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2; //1900.01.01
public static final int LATEST_DATE_ORDINAL = 2958465; //9999.12.31
```

Nie jest dla mnie jasne, dlaczego `EARLIEST_DATE_ORDINAL` ma wartość 2 zamiast 0. W wierszu 829. znajduje się sugestia, że ma to coś wspólnego ze sposobem reprezentacji dat w programie Microsoft Excel. Znacznie dokładniej jest to przedstawione w klasie `SpreadsheetDate` dziedziczącej po `DayDate` (listing B.5 w dodatku B). Komentarz z wiersza 71. dobrze opisuje problem.

Problem, jaki mamy tutaj, wydaje się związany z implementacją `SpreadsheetDate` i nie ma nic wspólnego z `DayDate`. Wywnioskowałem z tego, że `EARLIEST_DATE_ORDINAL` oraz `LATEST_DATE_ORDINAL` nie należą do `DayDate` i powinny zostać przeniesione do `SpreadsheetDate` [G6].

---

[2] Kilku recenzentów tego rozdziału protestowało przeciwko tej decyzji. Przekonywali oni, że w bibliotece open source lepiej zakładać ręczną kontrolę nad identyfikatorem serializacji, aby pomniejsze zmiany w oprogramowaniu nie powodowały unieważnienia starych serializowanych dat. Jest to zrozumiały powód. Jednak awarie, niezależnie od tego, jak mogą być niewygodne, mają jasną przyczynę. Z drugiej strony, jeżeli autor klasy zapomni zaktualizować identyfikator, to powód awarii nie jest zdefiniowany i może być dobrze ukryty. Uważam, że morałem z tej historii jest to, że nie powinniśmy oczekiwać deserializacji pomiędzy różnymi wersjami.

Faktycznie, przeszukanie kodu wykazuje, że zmienne te są używane wyłącznie w SpreadsheetDate. Nic w DayDate ani w innych klasach biblioteki JCommon nie korzysta z tych zmiennych. Dlatego przeniosłem je do SpreadsheetDate.

Następne zmienne, MINIMUM_YEAR_SUPPORTED oraz MAXIMUM_YEAR_SUPPORTED (wiersz 104. oraz 107.), spowodowały pewien dylemat. Wydaje się jasne, że jeżeli klasa DayDate jest klasą abstrakcyjną, która nie zapewnia implementacji, to nie powinna informować nas o roku minimalnym i maksymalnym. Kolejny raz miałem zamiar przenieść te zmienne do SpreadsheetDate [G6]. Jednak wyszukanie użytkowników tych zmiennych wykazało, że inna klasa z nich korzysta: RelativeDayOfWeekRule (listing B.6 w dodatku B). Są one zastosowane w wierszu 177. i 178. w funkcji getDate, gdzie są wykorzystywane do sprawdzenia, czy argument getDate ma prawidłowy rok. Dylemat jest związany z tym, że użytkownik klasy abstrakcyjnej potrzebuje informacji o jej implementacji.

Do dostarczenia tych informacji bez zanieczyszczania potrzebujemy klasy DayDate. Zwykle powinniśmy uzyskać informacje o implementacji z obiektu klasy pochodnej. Jednak do funkcji getDate nie jest przekazywany obiekt DayDate. Zwraca ona jednak taki obiekt, co oznacza, że musi być gdzieś utworzony. Podpowiedzią są wiersze od 187. do 205. Obiekt DayDate jest tworzony przez jedną z trzech funkcji, getPreviousDayOfWeek, getNearestDayOfWeek lub getFollowingDayOfWeek. Wracając do listingu DayDate, możemy zauważyć, że te funkcje (wiersze od 638. do 724.) zwracają datę utworzoną przez addDays (wiersz 571.), a ta wywołuje createInstance (wiersz 808.), która z kolei tworzy obiekt SpreadsheetDate! [G7].

Zwykle złym pomysłem jest, gdy klasy bazowe wiedzą o swoich klasach pochodnych. Aby to poprawić, powinniśmy użyć wzorca fabryki abstrakcyjnej[3] i utworzyć DayDateFactory. Fabryka ta utworzy potrzebne nam instancje DayDate i dodatkowo odpowie na pytania o implementację, takie jak data minimalna i maksymalna.

```
public abstract class DayDateFactory {
  private static DayDateFactory factory = new SpreadsheetDateFactory();
  public static void setInstance(DayDateFactory factory) {
    DayDateFactory.factory = factory;
  }

  protected abstract DayDate _makeDate(int ordinal);
  protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);
  protected abstract DayDate _makeDate(int day, int month, int year);
  protected abstract DayDate _makeDate(java.util.Date date);
  protected abstract int _getMinimumYear();
  protected abstract int _getMaximumYear();

  public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
  }

  public static DayDate makeDate(int day, DayDate.Month month, int year) {
    return factory._makeDate(day, month, year);
  }
```

[3] [GOF].

```
  public static DayDate makeDate(int day, int month, int year) {
    return factory._makeDate(day, month, year);
  }

  public static DayDate makeDate(java.util.Date date) {
    return factory._makeDate(date);
  }

  public static int getMinimumYear() {
    return factory._getMinimumYear();
  }

  public static int getMaximumYear() {
    return factory._getMaximumYear();
  }
}
```

Ta klasa fabryki zastępuje metody `createInstance` za pomocą metod `makeDate`, co nieco poprawia nazewnictwo [N1]. Domyślnie wykorzystywana jest klasa `SpreadsheetDateFactory`, ale może to być zmienione w dowolnym momencie przez użycie innej fabryki. Metody statyczne, które delegują do metod abstrakcyjnych, korzystają z połączenia wzorców: singleton[4], dekorator[5] i fabryka abstrakcyjna (moim zdaniem, wzorców bardzo użytecznych).

Kod `SpreadsheetDateFactory` wygląda następująco:

```
public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }

  public DayDate _makeDate(int day, DayDate.Month month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
      calendar.get(Calendar.DATE),
      DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
      calendar.get(Calendar.YEAR));
  }

  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }

  protected int _getMaximumYear() {
    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
  }
}
```

---

[4] Ibid.

[5] Ibid.

Jak można zauważyć, przeniosłem już zmienne MINIMUM_YEAR_SUPPORTED oraz MAXIMUM_YEAR_
↪SUPPORTED do SpreadsheetDate, do której to klasy one faktycznie należą [G6].

Następnym problemem w DayDate są stałe dni rozpoczynające się w wierszu 109. Powinny być one osobnym typem wyliczeniowym [J3]. Przedstawiałem już ten wzorzec, więc nie będę się tutaj powtarzał. Można go zobaczyć w końcowym listingu.

Następnie mamy serię zmiennych zaczynających się od LAST_DAY_OF_MONTH w wierszu 140. Moim pierwszym zastrzeżeniem było to, że opisujące je komentarze są nadmiarowe [C3]. Ich nazwy są wystarczające. Dlatego usunę te komentarze.

Wydaje się, że nie ma powodu, aby ta tabela nie była prywatna [G8], ponieważ dostępna jest funkcja statyczna lastDayOfMonth, która dostarcza tych samych danych.

Następna tabela, AGGREGATE_DAYS_TO_END_OF_MONTH, jest nieco tajemnicza, ponieważ nie jest używana w bibliotece JCommon [G9]. Dlatego ją usunąłem.

To samo dotyczy LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH.

Następna tabela, AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH, jest używana wyłącznie w SpreadsheetDate (wiersz 434. oraz 473.). Powstaje pytanie, czy nie powinna zostać przeniesiona do SpreadsheetDate. Argument za nieprzenoszeniem jej jest taki, że tabela ta nie jest specyficzna dla żadnej określonej implementacji [G6]. Z drugiej strony, nie istnieje inna implementacja poza SpreadsheetDate, więc tabela powinna zostać przeniesiona możliwie blisko miejsca użycia [G10].

Najlepszym dla mnie argumentem jest konieczność zachowania spójności [G11], więc powinniśmy oznaczyć tę tabelę jako prywatną i udostępnić ją za pomocą funkcji, na przykład julianDateOf
↪LastDayOfMonth. Wydaje się, że nikt nie potrzebuje tej funkcji. Co więcej, tabela może być łatwo przeniesiona do DayDate, jeżeli nowa implementacja tej klasy będzie jej potrzebowała. Dlatego ją przeniosłem.

To samo dotyczy LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH.

Następnie mamy trzy zestawy stałych, które mogą być zmienione na typy wyliczeniowe (wiersze od 162. do 205.). Pierwszy z nich pozwala na wybranie tygodnia w miesiącu. Zamieniłem go na typ wyliczeniowy o nazwie WeekInMonth.

```
public enum WeekInMonth {
  FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
  public final int index;

  WeekInMonth(int index) {
    this.index = index;
  }
}
```

Drugi zestaw stałych (wiersze 177. – 187.) jest nieco brzydszy. Stałe INCLUDE_NONE, INCLUDE_FIRST, INCLUDE_SECOND oraz INCLUDE_BOTH są wykorzystywane do opisywania, czy zdefiniowane daty końcowe zakresu powinny być dołączone do tego zakresu. Matematycznie rzecz ujmując, opisują

terminy „otwarty okres", „półotwarty okres" oraz „zamknięty okres". Uważam, że jaśniejsze będzie użycie nomenklatury matematycznej [N3], więc zmieniłem je na typ wyliczeniowy o nazwie `DateInterval` ze stałymi `CLOSED`, `CLOSED_LEFT`, `CLOSED_RIGHT` oraz `OPEN`.

Trzeci zbiór stałych (wiersze 189. – 205.) opisuje, czy wyszukiwanie określonego dnia tygodnia powinno dać w wyniku ostatnie, następne lub najbliższe wystąpienie. Wybór tej nazwy jest co najmniej trudny. W końcu zdecydowałem się na `WeekdayRange` ze stałymi `LAST`, `NEXT` oraz `NEAREST`.

Niekoniecznie można zgadzać się z wybranymi przeze mnie nazwami. Mają one dla mnie sens, ale niekoniecznie mogą mieć sens dla innych. Jednak są one teraz w takiej postaci, którą można bardzo łatwo zmieniać [J3]. Nigdzie nie są przekazywane w postaci liczb — są przekazywane jako symbole. Można użyć funkcji IDE „zmień nazwę" do zmiany nazwy lub korzystać z nich bez obawy, że w innym miejscu kodu opuściliśmy `-1` lub `2` albo że któraś z deklaracji argumentu `int` została źle opisana.

Pole opisowe z wiersza 208. nie jest już nigdzie używane. Usunąłem je więc razem z jego akcesorem i mutatorem.

Dodatkowo usunąłem zdegenerowany konstruktor domyślny z wiersza 213. [G12]. Kompilator wygeneruje go za nas.

Możemy pominąć metodę `isValidWeekdayCode` (wiersze 216. – 238.), ponieważ usunęliśmy ją podczas tworzenia typu wyliczeniowego `Day`.

Dotarliśmy więc do metody `stringToWeekdayCode` (wiersze 242. – 270.). Komentarze Javadoc, które nie dodają informacji do sygnatury metody, są jedynie wypełniaczem [C3], [G12]. Jedyną wartością tego komentarza Javadoc jest opis zwracanej wartości `-1`. Jednak ponieważ skorzystaliśmy z typu wyliczeniowego `Day`, komentarz ten jest teraz nieprawidłowy [C2]. Obecnie metoda zgłasza wyjątek `IllegalArgumentException`. Usunąłem więc Javadoc.

Usunąłem również słowa kluczowe `final` z deklaracji argumentów i zmiennych. Można powiedzieć, że nie dodają one żadnej rzeczywistej wartości, a jedynie zaciemniają kod [G12]. Eliminacja słów kluczowych `final` kłóci się z pewnymi konwencjami. Na przykład Robert Simmons[6] gorąco zachęca nas do „korzystania z `final` w całym kodzie". Nie zgadzam się z tym. Uważam, że istnieje trochę dobrych zastosowań `final`, takich jak niektóre stałe `final`, ale w innych przypadkach to słowo kluczowe niewiele daje poza zaśmiecaniem kodu. Być może uważam tak, ponieważ błędy, przed jakimi zabezpiecza słowo kluczowe `final`, mogą być wcześniej przechwycone przez pisane przeze mnie testy jednostkowe.

Nie podobają mi się powtórzone instrukcje `if` [G5] wewnątrz pętli `for` (wiersze 259. oraz 263.), więc połączyłem je za pomocą operatora `||` w jedną instrukcję `if`. Wykorzystałem również typ wyliczeniowy `Day` do sterowania pętlą i wprowadziłem kilka innych kosmetycznych zmian.

---

[6] [Simmons04], s. 73.

Zauważyłem również, że metoda ta nie należy tak naprawdę do `DayDate`. Jest to funkcja analizująca wartość `Day`. Dlatego przeniosłem ją do typu wyliczeniowego `Day`. Jednak spowodowało to, że typ `Day` stał się dosyć duży. Ponieważ zgodnie z koncepcją `Day` nie zależy od `DayDate`, przeniosłem typ wyliczeniowy `Day` poza klasę `DayDate`, do osobnego pliku źródłowego [G13].

Przeniosłem do niego również następną funkcję, `weekdayCodeToString` (wiersze 272. – 286.), i nazwałem ją `toString`.

```
public enum Day {
  MONDAY(Calendar.MONDAY),
  TUESDAY(Calendar.TUESDAY),
  WEDNESDAY(Calendar.WEDNESDAY),s
  THURSDAY(Calendar.THURSDAY),
  FRIDAY(Calendar.FRIDAY),
  SATURDAY(Calendar.SATURDAY),
  SUNDAY(Calendar.SUNDAY);

  public final int index;
  private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

  Day(int day) {
    index = day;
  }

  public static Day make(int index) throws IllegalArgumentException {
    for (Day d : Day.values())
      if (d.index == index)
        return d;
    throw new IllegalArgumentException(
      String.format("Niedozwolony indeks dnia: %d.", index));
  }

  public static Day parse(String s) throws IllegalArgumentException {
    String[] shortWeekdayNames =
      dateSymbols.getShortWeekdays();
    String[] weekDayNames =
      dateSymbols.getWeekdays();

    s = s.trim();
    for (Day day : Day.values()) {
      if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
          s.equalsIgnoreCase(weekDayNames[day.index])) {
        return day;
      }
    }
    throw new IllegalArgumentException(
      String.format("%s nie jest prawidłową nazwą dnia", s));
  }

  public String toString() {
    return dateSymbols.getWeekdays()[index];
  }
}
```

Następnie mamy dwie funkcje `getMonths` (wiersze 288. – 316.). Pierwsza wywołuje drugą. Druga nie jest wywoływana nigdzie poza pierwszą. Dlatego połączyłem te funkcje, znacznie je upraszczając [G9], [G12], [F4]. Na koniec zmieniłem nazwę na bardziej sugestywną [N1].

```
public static String[] getMonthNames() {
  return dateFormatSymbols.getMonths();
}
```

Funkcja `isValidMonthCode` (wiersze 326. – 346.) przestała być potrzebna przez zastosowanie typu wyliczeniowego `Month`, więc ją usunąłem [G9].

Metoda `monthCodeToQuarter` (wiersze 356. – 375.) wygląda na przypadek „zazdrości o funkcje”[7] (ang. *Feature Envy*) [G14] i prawdopodobnie należy do typu wyliczeniowego `Month` jako metoda `quarter`. Dlatego ją zastąpiłem.

```
public int quarter() {
  return 1 + (index-1)/3;
}
```

Spowodowało to, że typ wyliczeniowy `Month` stał się tak duży, iż powinien stać się osobną klasą. Przeniosłem go więc poza `DayDate`, zachowując spójność z typem wyliczeniowym `Day` [G11], [G13].

Następne dwie metody noszą nazwę `monthCodeToString` (wiersze 377. – 426.). Również w tym przypadku mamy wzorzec wywołania jednej metody z drugiej, z dodatkowo przekazanym znacznikiem. Zwykle przekazywanie znacznika jako argumentu funkcji jest złym pomysłem, szczególnie gdy znacznik wybiera format danych wyjściowych [G15]. Zmieniłem nazwę tej funkcji, zmieniłem strukturę i uprościłem ją, a następnie przeniosłem do typu wyliczeniowego `Month` [N1], [N3], [C3], [G14].

```
public String toString() {
  return dateFormatSymbols.getMonths()[index - 1];
}

public String toShortString() {
  return dateFormatSymbols.getShortMonths()[index - 1];
}
```

Kolejną metodą jest `stringToMonthCode` (wiersze 428. – 472.). Zmieniłem nazwę tej funkcji i uprościłem ją, a następnie przeniosłem do typu wyliczeniowego `Month` [N1], [N3], [C3], [G14], [G12].

```
public static Month parse(String s) {
  s = s.trim();
  for (Month m : Month.values())
    if (m.matches(s))
      return m;

  try {
    return make(Integer.parseInt(s));
  }
  catch (NumberFormatException e) {}
    throw new IllegalArgumentException("Niewłaściwy miesiąc " + s);
}

private boolean matches(String s) {
  return s.equalsIgnoreCase(toString()) ||
         s.equalsIgnoreCase(toShortString());
}
```

[7] [Refactoring].

Metoda `isLeapYear` (wiersze 495. – 513.) powinna być bardziej ekspresyjna [G16].

```
public static boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundredth = year % 100 == 0;
  boolean fourHundredth = year % 400 == 0;
  return fourth && (!hundredth || fourHundredth);
}
```

Następna funkcja, `leapYearCount` (wiersze 519. – 536.), nie należy do `DayDate`. Nic jej nie wywołuje poza dwoma metodami z `SpreadsheetDate`. Dlatego przeniosłem ją dalej.

Funkcja `lastDayOfMonth` (wiersze 538. – 560.) korzysta z tablicy `LAST_DAY_OF_MONTH`. Tablica należy naprawdę do typu wyliczeniowego `Month` [G17], więc ją tam przeniosłem. Dodatkowo uprościłem ją i zmieniłem na bardziej ekspresyjną [G16].

```
public static int lastDayOfMonth(Month month, int year) {
  if (month == Month.FEBRUARY && isLeapYear(year))
    return month.lastDay() + 1;
  else
    return month.lastDay();
}
```

Kolejne funkcje są bardziej interesujące. Następną funkcją jest `addDays` (wiersze 562. – 567.). Po pierwsze, ponieważ ta funkcja działa na zmiennych `DayDate`, nie powinna być statyczna [G18]. Dlatego zmieniłem ją na metodę instancyjną. Po drugie, wywołuje ona funkcję `toSerial`. Funkcja ta powinna zostać nazwana `toOrdinal` [N1]. Dodatkowo metoda ta może być uproszczona.

```
public DayDate addDays(int days) {
  return DayDateFactory.makeDate(toOrdinal() + days);
}
```

To samo dotyczy `addMonths` (wiersze 578. – 602.). Powinna to być metoda instancyjna [G18]. Algorytm jest dość skomplikowany, więc użyłem objaśniających zmiennych tymczasowych[8] [G19], aby był bardziej przejrzysty. Dodatkowo zmieniłem metodę `getYYY` na `getYear` [N1].

```
public DayDate addMonths(int months) {
  int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
  int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
  int resultYear = resultMonthAsOrdinal / 12;
  Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
  int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
  return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

W funkcji `addYears` (wiersze 604. – 626.) nie ma żadnych nowych niespodzianek.

```
public DayDate plusYears(int years) {
  int resultYear = getYear() + years;
  int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
  return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

[8] [Beck97].

Cichutki głosik w mojej głowie namawiał mnie na zmianę tych metod ze statycznych na instancyjne. Czy wyrażenie `date.addDays(5)` jasno pokazuje, że obiekt `date` nie zmienia się i zwracany jest nowy obiekt `DayDate`? Czy też błędnie przyjmujemy, że do obiektu `date` zostanie dodanych pięć dni? Można uważać, że nie jest to duży problem, ale poniższy fragment kodu pokazuje, że może to być bardzo mylące [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // Przesuwamy datę o tydzień.
```

Ktoś, kto czyta ten kod, najprawdopodobniej przyjmie, że `addDays` zmienia obiekt `date`. Musimy wybrać nazwę, która rozwieje tę niejasność [N4]. Zmieniłem więc nazwy na `plusDays` oraz `plusMonths`. Wydaje mi się, że przeznaczenie tej metody jest dobrze oddawane przez

```
DayDate date = oldDate.plusDays(5);
```

natomiast poniższe wywołanie nie da się przeczytać tak płynnie, aby czytelnik przyjął, że zmieniany jest obiekt `date`:

```
date.plusDays(5);
```

Algorytmy stają się coraz bardziej interesujące. Metoda `getPreviousDayOfWeek` (wiersze 628. – 660.) działa, ale jest skomplikowana. Po przeanalizowaniu, co naprawdę jest tu wykonywane [G21], byłem w stanie uprościć ją i użyć objaśniających zmiennych tymczasowych [G19], aby ją zapisać jaśniej. Zmieniłem ją również z metody statycznej na instancyjną [G18] i usunąłem powtórzoną metodę instancyjną [G5] (wiersze 997. – 1008.).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
  if (offsetToTarget >= 0)
    offsetToTarget -= 7;
  return plusDays(offsetToTarget);
}
```

Dokładnie taką samą analizę zastosowałem w `getFollowingDayOfWeek` (wiersze 662. – 693.), uzyskując analogiczne wyniki.

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
  if (offsetToTarget <= 0)
    offsetToTarget += 7;
  return plusDays(offsetToTarget);
}
```

Następną funkcją jest `getNearestDayOfWeek` (wiersze 695. – 726.), którą poprawiliśmy już wcześniej. Jednak wprowadzone zmiany nie są spójne z bieżącym wzorcem zastosowanym w ostatnich dwóch funkcjach [G11]. Aby uzyskać spójność, skorzystałem z kilku objaśniających zmiennych tymczasowych [G19] do wyjaśnienia algorytmu.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
  int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
  int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
  int offsetToPreviousTarget = offsetToFutureTarget - 7;

  if (offsetToFutureTarget > 3)
    return plusDays(offsetToPreviousTarget);
  else
    return plusDays(offsetToFutureTarget);
}
```

Metoda `getEndOfCurrentMonth` (wiersze 728. – 740.) jest dziwna, ponieważ jest to zmienna instancyjna, która „zazdrości" [G14] własnej klasie przez oczekiwanie argumentu `DayDate`. Zmieniłem ją na prawdziwą metodę instancyjną i rozjaśniłem kilka nazw.

```
public DayDate getEndOfMonth() {
  Month month = getMonth();
  int year = getYear();
  int lastDay = lastDayOfMonth(month, year);
  return DayDateFactory.makeDate(lastDay, month, year);
}
```

Przebudowa metody `weekInMonthToString` (wiersze 742. – 761.) okazała się jednak bardzo interesująca. Przy użyciu narzędzi środowiska IDE na początek przeniosłem metodę do typu wyliczeniowego `WeekInMonth`, który został utworzony nieco wcześniej. Następnie zmieniłem nazwę tej metody na `toString`. Potem zmieniłem ją z metody statycznej na instancyjną. Testy nadal były wykonywane prawidłowo (czy wiesz, do czego zmierzam?).

Następnie zupełnie ją usunąłem! Pięć asercji przestało działać (wiersze 411. – 415., listing B.4 w dodatku B). Zmieniłem te wiersze tak, aby wykorzystywane były nazwy stałych typu wyliczeniowego (`FIRST`, `SECOND`, …). Wszystkie testy były wykonywane prawidłowo. Czy możesz powiedzieć, dlaczego? Czy widzisz, dlaczego każdy z tych kroków był potrzebny? Narzędzie przebudowy kontroluje, czy wszystkie poprzednie wywołania `weekInMonthToString` obecnie są zamienione na `toString` ze stałych typu wyliczeniowego `weekInMonth`, ponieważ wszystkie stałe wyliczeniowe implementują metodę `toString`, która po prostu zwraca ich nazwę.

Niestety, było to zbyt proste. Niezależnie od tego, jak wspaniały był łańcuch przebudowy, ostatecznie zorientowałem się, że jedynymi użytkownikami tej funkcji były zmodyfikowane przeze mnie testy, więc je usunąłem.

Oszukasz mnie raz — wstydź się. Gdy oszukasz mnie po raz drugi, ja powinienem się wstydzić. Gdy zorientowałem się, że nic innego poza testami nie wywołuje metody `relativeToString` (wiersze 765. – 781.), po prostu usunąłem funkcję i jej testy.

W końcu dotarliśmy do metod abstrakcyjnych tej klasy abstrakcyjnej. Pierwsza jest najbardziej właściwa: `toSerial` (wiersze 838. – 844.). Na początku tego rozdziału zmieniłem jej nazwę na `toOrdinal`. Patrząc na jej kontekst, zdecydowałem się na zmianę jej nazwy na `getOrdinalDay`.

Następną metodą abstrakcyjną jest `toDate` (wiersze 838. – 844.). Konwertuje ona `DayDate` na `java.util.Date`. Dlaczego ta metoda jest abstrakcyjna? Jeżeli spojrzymy na implementację w `SpreadsheetDate` (wiersze 198. – 207., listing B.5 w dodatku B), okaże się, że nie zależy ona od niczego w implementacji tej klasy [G6]. Dlatego przesunąłem ją w górę.

Metody `getYYYY`, `getMonth` oraz `getDayOfMonth` tworzą ładną abstrakcję. Jednak metoda `gatDayOfWeek` jest kolejną, która powinna być wyjęta z `SpreadSheetDate`, ponieważ nie zależy od niczego innego, jak tylko od elementów znajdujących się w `DayDate` [G6]. A może nie?

Jeżeli przyjrzymy się jej dokładnie (wiersz 247., listing B.5 w dodatku B), zauważymy, że algorytm niejawnie zależy od położenia dni w kolejności (inaczej mówiąc, który dzień tygodnia ma indeks 0). Tak więc, mimo że ta funkcja nie ma fizycznych zależności, które nie mogą być przeniesione do `DayDate`, posiada zależności logiczne.

Tego typu zależności logiczne budzą moje wątpliwości [G22]. Jeżeli coś zależy logicznie od implementacji, to powinno również zależeć fizycznie. Ponadto wydaje mi się, że sam algorytm może być ogólny, a tylko niewielka jego część powinna być zależna od implementacji [G6].

Dlatego utworzyłem metodę abstrakcyjną w `DayDate`, o nazwie `getDayOfWeekForOrdinalZero`, i do `SpreadsheetDate` dodałem implementację zwracającą `Day.SATURDAY`. Następnie przeniosłem metodę `getDayOfWeek` do `DayDate` i zmieniłem ją, aby wywoływała `getOrdinalDay` oraz `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {
  Day startingDay = getDayOfWeekForOrdinalZero();
  int startingOffset = startingDay.index - Day.SUNDAY.index;
  return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Proponuję przyjrzeć się uważniej komentarzowi znajdującemu się w wierszach od 895. do 899. Czy takie powtórzenie jest naprawdę potrzebne? Jak zwykle, usunąłem ten komentarz wraz z innymi.

Następną metodą jest `compare` (wiersze 902. – 913.). Po raz kolejny metoda ta jest na niewłaściwym poziomie abstrakcji [G6], więc przeniosłem jej implementację do `DayDate`. Dodatkowo nazwa nie jest wystarczająco komunikatywna [N1]. Metoda faktycznie zwraca różnicę w dniach w stosunku do argumentu. Dlatego zmieniłem jej nazwę na `daysSince`. Zauważyłem też, że nie ma dla tej metody testów, więc je dopisałem.

Sześć kolejnych funkcji (wiersze od 915. do 980.) to funkcje abstrakcyjne, które powinny być implementowane w `DayDate`. Dlatego wyciągnąłem je z `SpreadsheetDate`.

Ostatnia funkcja, `isInRange` (wiersze 982. – 995.), również wymaga wydobycia i przebudowy. Instrukcja `switch` jest niezbyt elegancka [G23] i może być zamieniona przez przeniesienie przypadków do typu wyliczeniowego `DateInterval`.

```
public enum DateInterval {
  OPEN {
    public boolean isIn(int d, int left, int right) {
      return d > left && d < right;
    }
  },
  CLOSED_LEFT {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d < right;
    }
  },
  CLOSED_RIGHT {
    public boolean isIn(int d, int left, int right) {
      return d > left && d <= right;
    }
  },
  CLOSED {
```

```
    public boolean isIn(int d, int left, int right) {
      return d >= left && d <= right;
    }
  };
  public abstract boolean isIn(int d, int left, int right);
}


public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
  int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
  int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
  return interval.isIn(getOrdinalDay(), left, right);
}
```

W ten sposób dotarliśmy do końca klasy `DayDate`. Kolejnym krokiem będzie ponowne przejrzenie całej klasy i sprawdzenie, czy przepływ kodu jest właściwy.

Otwierający komentarz jest od dawna niewłaściwy, więc skróciłem go i poprawiłem [C2].

Następnie przeniosłem pozostałe typy wyliczeniowe do osobnych plików [G12].

Potem przeniosłem zmienną statyczną (`dateFormatSymbols`) oraz trzy metody statyczne (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) do nowej klasy o nazwie `DateUtil` [G6].

Przeniosłem metody abstrakcyjne w górę, tam gdzie powinny być [G24].

Zmieniłem `Month.make` na `Month.fromInt` [N1] i wykonałem to samo w przypadku pozostałych typów wyliczeniowych. Dodatkowo we wszystkich typach wyliczeniowych utworzyłem akcesor `toInt()` i pole `index` zmieniłem na prywatne.

Znalazłem również kilka interesujących powtórzeń [G5] w `plusYears` oraz `plusMonths`, które mogłem wyeliminować przez wyodrębnienie nowej metody o nazwie `correctLastDayOfMonth`, co spowodowało, że wszystkie trzy metody są znacznie bardziej czytelne.

Usunąłem również magiczną liczbę 1 [G25], zastępując ją wywołaniami, odpowiednio: `Month.JANU`
`↪ARY.toInt()` lub `Day.SUNDAY.toInt()`. Poświęciłem również trochę czasu klasie `SpreadsheetDate`, poprawiając algorytm. Końcowy wynik znajduje się na listingach od B.7 do B.16 w dodatku B.

Interesujące jest, że pokrycie kodu w `DayDate` *zmniejszyło się* do 84,9 procent. Stało się tak nie z powodu mniejszej liczby testowanych funkcji — po prostu klasa na tyle się zmniejszyła, że tych kilka niepokrytych wierszy ma większą wagę. `DayDate` ma obecnie pokrytych testami 45 z 53 instrukcji wykonywalnych. Niepokryte wiersze są tak proste, że nie warto ich testować.

Zakończenie

Kolejny raz zadziałaliśmy zgodnie z zasadą skautów. Zwróciliśmy kod nieco czyściejszy, niż go pobraliśmy. Zajęło to trochę czasu, ale się opłaciło. Pokrycie kodu testami zwiększyło się, zostało poprawionych kilka błędów, a kod został skrócony i jest bardziej zrozumiały. Mamy nadzieję, że następna osoba, która zajrzy do tego kodu, uzna, iż jest łatwiejszy w użyciu niż wcześniej. Mamy również nadzieję, że osoba ta usprawni kod w większym stopniu niż my.

Bibliografia

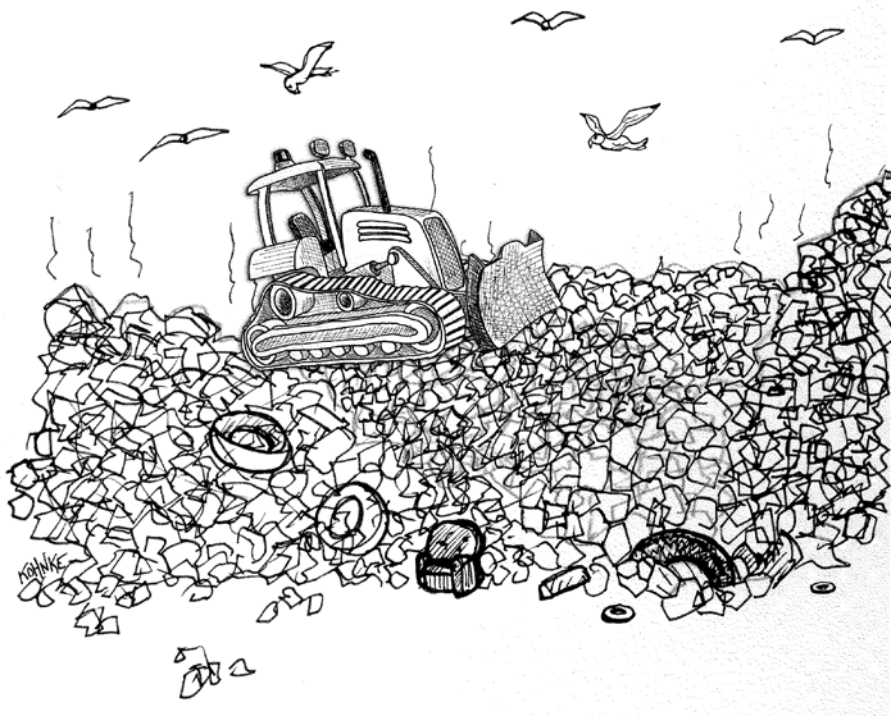
[GOF]: Gamma i inni, *Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

[Simmons04]: Robert Simmons Jr., *Hardcore Java*, O'Reilly 2004.

[Refactoring]: Martin Fowler i inni, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999.

[Beck97]: Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall 1997.

Zapachy kodu i heurystyki



W swojej wspaniałej książce *Refactoring*[1] Martin Fowler zidentyfikował wiele różnych „zapachów kodu". Na znajdującej się dalej liście znajduje się wiele zapachów zidentyfikowanych przez Martina i jeszcze więcej określonych przeze mnie. Dodatkowo dołączyłem inne perełki i heurystyki, którymi chciałbym się podzielić.

[1] [Refactoring].

Listę tę napisałem, gdy przeglądałem kilka różnych programów i przebudowywałem je. Wprowadzając zmianę, zadawałem sobie pytanie, *dlaczego* ją wprowadzam, a następnie zapisywałem powód. Wynikiem jest dosyć długa lista elementów, które brzydko mi pachną, gdy czytam kod źródłowy.

Lista ta może być czytana od początku do końca, jak również używana jako skorowidz. W dodatku C znajduje się lista odwołań każdej heurystyki, zawierająca wyszczególnienie miejsc w tekście, w których została użyta.

Komentarze

C1. Niewłaściwe informacje

Niewłaściwe jest, aby komentarze przechowywały informacje, które lepiej przechowywać w innych systemach, takich jak system kontroli wersji, system śledzenia błędów lub inny system przechowywania danych. Na przykład historia zmian jedynie zaciemnia pliki źródłowe wieloma wierszami historycznego i mało interesującego tekstu. Zwykle metadane, takie jak autorzy, data ostatniej modyfikacji, numer SPR itd., nie powinny występować w komentarzach. Komentarze powinny być zarezerwowane dla informacji technicznych o kodzie i projekcie.

C2. Przestarzałe komentarze

Komentarz, który stał się nieistotny i nieprawidłowy, jest przestarzały. Komentarze bardzo szybko się starzeją. Najlepiej nie pisać komentarzy, które mogą utracić aktualność. Jeżeli znajdziemy przestarzały komentarz, najlepiej jak najszybciej go zaktualizować lub usunąć. Przestarzałe komentarze mają tendencję do utraty łączności z opisywanym kodem. Stają się one pływającymi wyspami nieistotności i błędnymi kierunkowskazami w kodzie.

C3. Nadmiarowe komentarze

Komentarz jest nadmiarowy, jeżeli opisuje coś, co wystarczająco dobrze samo się opisuje. Na przykład:

```
i++; // Zwiększenie i
```

Innym przykładem jest Javadoc niezawierający więcej informacji (a nawet mniej) niż sygnatura funkcji:

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
  throws ManagedComponentException
```

Komentarze powinny informować o tym, o czym nie może powiedzieć sam kod.

C4. Źle napisane komentarze

Komentarz, który wart jest napisania, trzeba napisać dobrze. Jeżeli decydujemy się na napisanie komentarza, musimy poświęcić trochę czasu na upewnienie się, że jest to najlepszy komentarz, jaki mogliśmy napisać. Należy uważnie dobierać słowa, przestrzegać zasad odmiany i interpunkcji. Nie pisać chaotycznie! Nie pisać o oczywistościach. Pisać zwięźle.

C5. Zakomentowany kod

Do szaleństwa doprowadzają mnie połacie zakomentowanego kodu. Kto będzie wiedział, jak stary jest ten kod? Kto będzie wiedział, czy jest on znaczący, czy nie? Dodatkowo nikt go nie skasuje, ponieważ każdy zakłada, że ktoś inny go potrzebuje lub ma względem niego jakieś plany.

Komentarz taki tkwi w kodzie przez dłuższy czas, psując się i z każdym dniem tracąc na znaczeniu. Wywołuje funkcje, które już nie istnieją. Używa zmiennych, których nazwy zostały zmienione. Korzysta z konwencji, które już od dawna nie obowiązują. Zanieczyszcza zawierający go moduł i przeszkadza ludziom, którzy go czytają. Zakomentowany kod jest *potworny*.

Gdy zobaczysz zakomentowany kod, *usuń go!* Nie obawiaj się, system kontroli wersji nadal go pamięta. Jeżeli ktokolwiek będzie go potrzebował, będzie mógł pobrać wcześniejszą wersję. Nie toleruj zachowywania zakomentowanego kodu.

Środowisko

E1. Budowanie wymaga więcej niż jednego kroku

Budowanie (kompilacja i konsolidacja) projektu powinno być jedną prostą operacją. Niedopuszczalne jest pobieranie wielu elementów z systemu kontroli wersji. Nie powinniśmy potrzebować sekwencji zaawansowanych poleceń lub skryptów zależnych od kontekstu przeznaczonych do zbudowania poszczególnych elementów. Nie powinniśmy szukać dodatkowych plików JAR, XML i innych wymaganych przez system artefaktów. *Powinniśmy* mieć możliwość pobrania systemu za pomocą jednego prostego polecenia, a następnie wykonania innego prostego polecenia w celu jego zbudowania.

```
svn get mySystem
cd mySystem
ant all
```

E2. Testy wymagają więcej niż jednego kroku

Powinniśmy być w stanie uruchomić *wszystkie* testy jednostkowe za pomocą jednego polecenia. Najlepszym przypadkiem jest uruchomienie wszystkich testów kliknięciem jednego przycisku w IDE. W najgorszym przypadku powinniśmy to osiągnąć, wywołując jedno proste polecenie powłoki. Możliwość wykonania wszystkich testów jest tak ważna i podstawowa, że powinna być szybka, łatwa i oczywista.

Funkcje

F1. Nadmiar argumentów

Funkcje powinny mieć małą liczbę argumentów. Najlepszym przypadkiem jest... brak argumentów, następnie mogą być: jeden, dwa lub trzy argumenty. Większa liczba argumentów stanowi potencjalne źródło problemów (patrz „Argumenty funkcji" w rozdziale 3.).

F2. Argumenty wyjściowe

Argumenty wyjściowe są mało intuicyjne. Czytelnik oczekuje, że argumenty będą wejściem, a nie wyjściem. Jeżeli funkcja musi zmieniać stan czegokolwiek, powinna zmieniać stan własnego obiektu (patrz „Argumenty wyjściowe" w rozdziale 3.).

F3. Argumenty znacznikowe

Argumenty logiczne deklarują, że funkcja wykonuje więcej niż jedną operację. Są one mylące i powinny być eliminowane (patrz „Argumenty znacznikowe" w rozdziale 3.).

F4. Martwe funkcje

Metody, które nie są nigdy wywoływane, powinny być usuwane. Przechowywanie martwego kodu jest rozrzutnością. Nie trzeba się obawiać usuwania funkcji — system kontroli wersji nadal ją pamięta.

Ogólne

G1. Wiele języków w jednym pliku źródłowym

Stosowane obecnie środowiska programowania umożliwiają umieszczanie wielu różnych języków w jednym pliku źródłowym. Na przykład plik źródłowy języka Java może zawierać fragmenty języków: XML, HTML, YAML, JavaDoc, polskiego, JavaScript i tak dalej. W przypadku plików JSP oprócz języka HTML można znaleźć kod Java, znaczniki biblioteki, komentarze w języku naturalnym, Javadoc, XML, JavaScript itd. Jest to co najmniej mylące.

W idealnym przypadku jeden plik źródłowy powinien zawierać jeden i tylko jeden język. Realistycznie rzecz biorąc, konieczne będzie użycie więcej niż jednego. Jednak powinniśmy podjąć próbę ograniczenia liczby i zakresu dodatkowych języków w plikach źródłowych.

G2. Oczywiste działanie jest nieimplementowane

Zgodnie z „zasadą najmniejszego zaskoczenia[2]" każda funkcja lub klasa powinna być implementowana w taki sposób, aby inny programista mógł się tego spodziewać. Przyjrzyjmy się funkcji przekształcającej nazwę dnia na typ `enum` reprezentujący ten dzień:

```
Day day = DayDate.StringToDay(String dayName);
```

Możemy oczekiwać, że ciąg *Monday* zostanie zamieniony na `Day.MONDAY`. Możemy również oczekiwać, że przekształcane będą często stosowane skróty, a funkcja będzie ignorowała wielkość liter.

Gdy takie oczywiste działania nie są implementowane, czytelnicy i użytkownicy kodu nie mogą polegać na swojej intuicji i nazwach funkcji. Tracą wówczas zaufanie do autora i muszą wrócić do czytania szczegółów kodu.

G3. Niewłaściwe działanie w warunkach granicznych

Wydaje się oczywiste, że kod powinien działać prawidłowo. Niestety, rzadko zdajemy sobie sprawę, jak skomplikowane jest prawidłowe działanie. Programiści często piszą funkcje, które według nich działają; ufają oni swojej intuicji, zamiast udowodnić, że kod działa we wszystkich sytuacjach skrajnych.

Niczym nie można zastąpić dokładnej analizy. Każdy warunek graniczny, każdy problem i wyjątek reprezentuje coś, co może zepsuć elegancki i intuicyjny algorytm. *Nie polegaj na swojej intuicji*. Należy wyszukać wszystkie warunki graniczne i napisać dla nich testy.

G4. Zdjęte zabezpieczenia

Do wybuchu w Czarnobylu doszło dlatego, że dyrektor elektrowni wyłączał kolejne mechanizmy zabezpieczające. Zabezpieczenia te utrudniały przeprowadzanie eksperymentu. Eksperyment i tak się nie udał, a świat zobaczył pierwszą dużą cywilną katastrofę nuklearną.

Zdejmowanie zabezpieczeń jest ryzykowne. Przejmowanie ręcznej kontroli nad wartością `serialVersionUID` może być niezbędne, ale zawsze jest ryzykowne. Wyłączanie określonych ostrzeżeń kompilatora (lub wszystkich ostrzeżeń!) może pomóc nam w prawidłowym skompilowaniu, ale niesie ryzyko nieskończonej sesji debugowania. Wyłączenie niewykonywanych testów i obiecanie sobie, że zajmiemy się nimi później, jest równie złe, jak postrzeganie kart kredytowych jako niewyczerpanego źródła darmowych pieniędzy.

---

[2] Lub „zasadą najmniejszego zdziwienia": *http://en.wikipedia.org/wiki/Principle_of_least_astonishment*.

G5. Powtórzenia

Jest to jedna z najważniejszych zasad z tej książki i powinniśmy ją traktować bardzo poważnie. Niemal każdy autor piszący o projektowaniu oprogramowania wspomina o tej zasadzie. Dave Thomas oraz Andy Hunt nazwali ją zasadą DRY[3] (ang. *Don't Repeat Yourself*), czyli „nie powtarzaj się". Kent Beck uznał ją za jedną z podstawowych zasad programowania ekstremalnego i nazwał ją „once, and only once", czyli „raz i tylko raz". Ron Jeffries uznał ją za drugą w hierarchii ważności, zaraz za wykonywaniem wszystkich testów.

Za każdym razem, gdy widzimy powtórzenie w kodzie, reprezentuje ono utraconą możliwość abstrakcji. Powtórzenie takie może prawdopodobnie stać się funkcją albo zupełnie osobną klasą. Przez zamianę powtórzeń na takie abstrakcje zwiększamy słownik projektowanego przez nas języka. Inni programiści mogą skorzystać z tworzonych przez nas abstrakcji. Kodowanie staje się szybsze i bardziej odporne na błędy, ponieważ podwyższamy poziom abstrakcji.

Najbardziej oczywistą formą powtórzenia jest występowanie fragmentów identycznego kodu, co wygląda, jakby któryś z programistów oszalał i wielokrotnie wklejał ten sam kod. Powinny być one zastąpione prostymi metodami.

Bardziej subtelną formą jest łańcuch instrukcji `switch-case` lub `if-else`, który występuje wielokrotnie w różnych modułach i zawsze używany jest ten sam zbiór warunków. Powinny być one zastąpione polimorfizmem.

Jeszcze bardziej subtelnymi powtórzeniami są moduły, które mają podobne algorytmy, ale nie mają podobnego kodu. Jest to również powtórzenie, które powinno być wyeliminowane przez użycie wzorców szablonu metody[4] lub strategii[5].

Faktycznie, większość wzorców projektowych, jakie zostały zdefiniowane w ostatnich piętnastu latach, to dobrze znane sposoby na eliminowanie powtórzeń. Również postacie normalne Codda są strategią eliminowania powtórzeń ze schematu bazy danych. Samo programowanie obiektowe jest strategią organizacji modułów i eliminowania powtórzeń. Nie dziwi zatem fakt, że to samo dotyczy programowania strukturalnego.

Uważam, że zasada została ustanowiona. Znajdź i wyeliminuj wszystkie powtórzenia.

G6. Kod na nieodpowiednim poziomie abstrakcji

Ważne jest, aby tworzyć abstrakcje, które oddzielają ogólne koncepcje na wyższym poziomie od szczegółów na niższym poziomie. Czasami robimy to przez tworzenie klas abstrakcyjnych, które przechowują koncepcje wyższego poziomu, a klasy dziedziczące po nich zawierają koncepcje niższych poziomów. Gdy stosujemy taki podział, musimy upewnić się, że jest kompletny. Chcemy,

---

[3] [PRAG].

[4] [GOF].

[5] [GOF].

aby *wszystkie* koncepcje niższych poziomów znajdowały się w klasach pochodnych, a *wszystkie* koncepcje wyższego poziomu były w klasie bazowej.

Na przykład stałe, zmienne lub funkcje użytkowe, które odnoszą się tylko do szczegółowej implementacji, nie powinny znajdować się w klasie bazowej. Klasa bazowa nie powinna nic o nich wiedzieć.

Zasada ta dotyczy również plików źródłowych, komponentów i modułów. Dobry projekt oprogramowania wymaga, abyśmy rozdzielali koncepcje na różne poziomy i umieszczali je w różnych kontenerach. Czasami kontenery te są klasami bazowymi lub pochodnymi, a czasami są plikami źródłowymi, modułami lub komponentami. Niezależnie od zastosowanego przypadku, rozdzielenie powinno być pełne. Nie powinniśmy mieszać ze sobą koncepcji z niższych i wyższych poziomów.

Weźmy pod uwagę następujący kod:

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();
  class EmptyException extends Exception {}
  class FullException extends Exception {}
}
```

Funkcja `percentFull` znajduje się na niewłaściwym poziomie abstrakcji. Choć istnieje wiele implementacji `Stack`, w których koncepcja *wypełnienia* jest rozsądna, to jednak istnieją inne implementacje, które po prostu *nie powinny wiedzieć*, jak bardzo są wypełnione. Dlatego funkcja ta powinna być umieszczona w interfejsie dziedziczącym, na przykład `BoundedStack`.

Ktoś może uważać, że taka implementacja powinna zwracać zero, jeżeli stos jest bez ograniczeń. Problem polega jednak na tym, że żaden stos nie jest naprawdę nieograniczony. Nie można naprawdę zapobiec wyjątkom `OutOfMemoryException` przez następujące sprawdzenia:

```
stack.percentFull() < 50.0.
```

Implementacja funkcji, która zwraca `0`, będzie po prostu wprowadzała w błąd.

Nie można wprowadzać w błąd lub maskować źle umieszczonej abstrakcji. Izolacja abstrakcji jest jednym z najtrudniejszych zadań programistów i w przypadku, gdy zostanie przeprowadzona źle, nie ma szybkiego sposobu poprawienia kodu.

G7. Klasy bazowe zależne od swoich klas pochodnych

Najczęstszym powodem podziału koncepcji na klasę bazową i pochodne jest to, że klasa bazowa wyższego poziomu może być niezależna od klas pochodnych niższego poziomu. Dlatego, gdy widzimy klasę bazową korzystającą z nazw swoich pochodnych, oczekujemy problemu. Klasy bazowe nie powinny nic wiedzieć o swoich potomkach.

Istnieją oczywiście wyjątki od tej zasady. Czasami liczba klas pochodnych jest ściśle ustalona i klasa bazowa posiada kod wybierający pomiędzy tymi klasami. Widzieliśmy takie rozwiązanie w wielu implementacjach automatów skończonych. Jednak w takim przypadku klasy pochodne i klasa bazowa

są ściśle połączone i zawsze udostępniane razem, w tym samym pliku *jar*. W ogólnym przypadku chcemy mieć możliwość udostępniania klas bazowych i pochodnych w różnych plikach *jar*.

Udostępnianie klas pochodnych w różnych plikach *jar* i zapewnienie, że bazowe pliki *jar* nic nie wiedzą o zawartości plików *jar* z klasami pochodnymi, pozwala nam na instalowanie systemów w indywidualnych i niezależnych komponentach. Gdy taki komponent zostanie zmodyfikowany, może być ponownie zainstalowany, bez konieczności instalowania komponentów bazowych. Oznacza to, że wpływ zmian jest znacznie obniżony, a utrzymanie działających systemów jest znacznie prostsze.

G8. Za dużo informacji

Dobrze zdefiniowane moduły mają bardzo małe interfejsy, które umożliwiają wykonanie wielu operacji. Źle zdefiniowane moduły mają szerokie i głębokie interfejsy, które wymuszają na nas tworzenie wielu różnych konstrukcji do wykonania prostych operacji. Dobrze zdefiniowane interfejsy nie posiadają wielu funkcji, od których zależą, więc sprzężenie jest niskie. Źle zdefiniowane interfejsy mają wiele funkcji, które należy wywołać, więc sprzężenie jest wysokie.

Profesjonalni twórcy oprogramowania ograniczają liczbę elementów udostępnianych w interfejsach ich klas i modułów. Im mniej metod ma klasa, tym lepiej. Im mniej zmiennych jest wykorzystywanych w funkcji, tym lepiej. Im mniej zmiennych instancyjnych ma klasa, tym lepiej.

Ukrywajmy nasze dane. Ukrywajmy funkcje narzędziowe. Ukrywajmy stałe i dane tymczasowe. Nie twórzmy klas z wieloma metodami i wieloma zmiennymi instancyjnymi. Nie twórzmy wielu zmiennych i funkcji zabezpieczonych dla naszych klas bazowych. Skoncentrujmy się na zachowaniu małych i ścisłych interfejsów. Eliminujmy sprzężenia przez ograniczanie informacji.

G9. Martwy kod

Martwy kod to taki, który nie jest wykonywany. Można znaleźć go w treści instrukcji `if`, które kontrolują nigdy niespełniane warunki. Można znaleźć go w blokach `catch`, dla których nigdy nie zostanie wywołane odpowiednie `throw`. Można go znaleźć w niewielkich metodach narzędziowych, które nie są nigdy wywoływane, lub w warunkach `switch-case`, które nigdy nie zostają spełnione.

Problem z martwym kodem jest taki, że szybko zaczyna wydzielać zapach. Im jest starszy, tym ten zapach staje się silniejszy i kwaśniejszy. Dzieje się tak, ponieważ martwy kod nie jest w pełni aktualizowany w czasie zmian projektu. Nadal *kompiluje się*, ale nie jest zgodny z najnowszymi konwencjami i zasadami. Był on pisany w czasie, gdy system był *inny*. Gdy znajdziemy martwy kod, możemy zrobić tylko jedną właściwą rzecz. Zapewnijmy mu godny pochówek, usuwając go z systemu.

G10. Separacja pionowa

Zmienne i funkcje powinny być definiowane blisko miejsca, gdzie są użyte. Zmienne lokalne powinny być deklarowane bezpośrednio ponad pierwszym użyciem i powinny mieć mały zakres pionowy. Nie chcemy zmiennych lokalnych deklarowanych setki wierszy od ich zastosowania.

Funkcje prywatne powinny być definiowane poniżej pierwszego zastosowania. Funkcje prywatne należą do zakresu całej klasy, ale i tak chcemy ograniczyć odległość w pionie pomiędzy wywołaniem a definicją. Znalezienie funkcji prywatnej powinno sprowadzać się do przewinięcia kodu w dół od pierwszego użycia.

G11. Niespójność

Jeżeli wykonujemy coś w określony sposób, wszystkie podobne zadania powinniśmy wykonywać w taki sam sposób. Odnosi się to do zasady najmniejszego zaskoczenia. Należy rozważnie wybierać konwencje i po ich wybraniu konsekwentnie je stosować.

Jeżeli w określonej funkcji korzystamy ze zmiennej o nazwie `response`, zawierającej `HttpServlet ↪Response`, to tej samej nazwy zmiennej powinniśmy używać we wszystkich innych funkcjach korzystających z obiektu `HttpServletResponse`. Jeżeli nazwiemy metodę `processVerificationRequest`, to powinniśmy użyć podobnej nazwy, takiej jak `processDeletionRequest`, dla metod przetwarzających inne rodzaje żądań.

Tego typu prosta spójność, jeżeli jest konsekwentnie stosowana, może spowodować, że kod będzie znacznie łatwiejszy do czytania i modyfikacji.

G12. Zaciemnianie

Jakie jest zastosowanie domyślnego konstruktora bez implementacji? Służy on jedynie do zaciemniania kodu fragmentami niemającymi znaczenia. Zmienne, które nie są używane, nigdy niewywoływane funkcje, komentarze, które nie dodają żadnej informacji i tym podobne elementy zaciemniają kod i powinny zostać usunięte. Pliki źródłowe winny być uporządkowane, dobrze zorganizowane.

G13. Sztuczne sprzężenia

Elementy, które nie są od siebie zależne, nie powinny być sztucznie łączone. Na przykład ogólny typ `enum` nie powinien znajdować się w bardziej specjalizowanej klasie, ponieważ wymusza to na całej aplikacji korzystanie ze zbyt szczegółowych klas. To samo dotyczy funkcji statycznej ogólnego przeznaczenia zadeklarowanej w specjalizowanej klasie.

Sztuczne sprzężenie to takie, które występuje pomiędzy dwoma niepowiązanymi bezpośrednio modułami. Jest to wynik umieszczenia zmiennej, stałej lub funkcji w tymczasowo wygodnej, choć niewłaściwej lokalizacji. Jest to wynik lenistwa i niedbalstwa.

Należy poświęcić trochę czasu na określenie, w którym miejscu powinny być zadeklarowane nasze funkcje, stałe i zmienne. Nie należy ich po prostu wrzucać w najwygodniejsze w danej chwili miejsce i pozostawiać ich tam.

G14. Zazdrość o funkcje

Jest to jeden z zapachów kodu[6] Martina Fowlera. Klasa (lub metoda) powinna być zainteresowana zmiennymi i funkcjami należącymi do tej klasy, a nie zmiennymi i metodami innych klas. Gdy metoda korzysta z akcesorów i mutatorów innego obiektu do manipulowania danymi wewnątrz tego obiektu, to *zazdrości* zakresu klasy tego obiektu. Żałuje, że nie znajduje się wewnątrz innej klasy, by mieć bezpośredni dostęp do zmiennych, którymi manipuluje. Na przykład:

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

Metoda `calculateWeeklyPay` sięga do obiektu `HourlyEmployee` w celu uzyskania danych, na których operuje. Metoda `calculateWeeklyPay` *zazdrości* zakresu `HourlyEmployee`. „Żałuje" ona, że nie jest wewnątrz `HourlyEmployee`.

Eliminujemy z kodu zazdrość o funkcje, ponieważ powoduje to ujawnienie jednej klasie wewnętrznej budowy innej klasy. Czasami jednak zazdrość o funkcje jest złem koniecznym. Weźmy pod uwagę następujący kod:

```
public class HourlyEmployeeReport {
  private HourlyEmployee employee ;

  public HourlyEmployeeReport(HourlyEmployee e) {
    this.employee = e;
  }

  String reportHours() {
    return String.format(
      "Nazwisko: %s\tGodziny:%d.%1d\n",
      employee.getName(),
      employee.getTenthsWorked()/10,
      employee.getTenthsWorked()%10);
  }
}
```

Oczywiście, metoda `reportHours` zazdrości klasie `HourlyEmployee`. Z drugiej strony, nie chcemy, aby klasa `HourlyEmployee` znała szczegóły formatowania raportu. Przeniesienie tego ciągu formatującego do klasy `HourlyEmployee` złamałoby kilka zasad projektowania obiektowego[7]. Spowodowałoby sprzężenie `HourlyEmployee` z formatem raportu, wystawiając ją na zmiany w formacie.

---

[6] [Refactoring].

[7] Dokładnie — zasadę pojedynczej odpowiedzialności, zasadę otwarty-zamknięty oraz zasadę wspólnego zamknięcia. Patrz [PPP].

G15. Argumenty wybierające

Trudno o coś bardziej wstrętnego, jak wiszący argument `false` na końcu wywołania funkcji. Co on oznacza? Co się stanie, jeżeli zmienimy go na `true`? Nie tylko trudno zapamiętać przeznaczenie argumentu wybierającego, ale również każdy argument wybierający łączy kilka funkcji w jednej. Argumenty wybierające są sposobem na uniknięcie podziału dużej funkcji na kilka mniejszych, wynikającym z lenistwa. Przeanalizujmy taki przykład:

```
public int calculateWeeklyPay(boolean overtime) {
  int tenthRate = getTenthRate();
  int tenthsWorked = getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
  int straightPay = straightTime * tenthRate;
  double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
  int overtimePay = (int)Math.round(overTime*overtimeRate);
  return straightPay + overtimePay;
}
```

Można wywołać tę funkcję z wartością `true`, jeżeli nadgodziny są płatne ze współczynnikiem `1,5`, lub `false`, jeżeli są płacone w zwykły sposób. Niewłaściwe jest, że musimy pamiętać, co oznacza wywołanie `calculateWeeklyPay(false)`, gdy się na takie natkniemy. Jednak prawdziwym wstydem jest to, że autor nie korzysta z możliwości napisania następującego kodu:

```
public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

Oczywiście, argument wybierający nie musi być typu `boolean`. Może być to typ wyliczeniowy, całkowity lub inny, który może być użyty do wyboru zachowania funkcji. Zwykle lepiej mieć więcej funkcji, niż przekazywać pewien kod do funkcji w celu wybrania zachowania.

G16. Zaciemnianie intencji

Kod powinien być wyrazisty i zrozumiały. Rozwlekłe wyrażenia, notacja węgierska i magiczne liczby zaciemniają intencje autora. Mamy tu na przykład funkcję `overTimePay`:

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
    (int) Math.round(0.5 * iThsRte *
            Math.max(0, iThsWkd - 400)
          );
}
```

Wygląda ona na małą i zwięzłą, ale jest kompletnie enigmatyczna. Warto poświęcić trochę czasu na ujawnienie czytelnikom zamierzeń naszego kodu.

G17. Źle rozmieszczona odpowiedzialność

Jedną z najważniejszych decyzji twórcy oprogramowania jest rozmieszczenie kodu. Gdzie na przykład powinna znaleźć się stała `PI`? Czy powinna być w klasie `Math`? Być może należy do klasy `Trigonometry`? A może do klasy `Circle`?

Ma tu zastosowanie zasada najmniejszego zaskoczenia. Kod powinien być umieszczony tam, gdzie spodziewa się go czytelnik. Stała `PI` powinna znaleźć się tam, gdzie są zadeklarowane funkcje trygonometryczne. Stała `OVERTIME_RATE` powinna znajdować się w klasie `HourlyPay` ↪`Calculator`.

Czasami chcemy zastosować „sprytne" rozwiązanie określonych funkcji. Umieszczamy funkcję tam, gdzie jest nam wygodnie, nie biorąc pod uwagę oczekiwań czytelnika. Możemy na przykład potrzebować raportu z sumą godzin przepracowanych przez pracownika. Możemy zsumować te godziny w kodzie drukującym raport albo spróbować tworzyć sumę w kodzie akceptującym karty czasu pracy.

Decyzję tę będzie nam łatwiej podjąć, gdy spojrzymy na nazwy funkcji. Załóżmy, że nasz moduł raportowy posiada funkcję o nazwie `getTotalHours`. Załóżmy również, że moduł akceptacji kart czasu pracy posiada funkcję `saveTimeCard`. Która nazwa tych dwóch funkcji informuje nas, że jest w niej obliczana sumaryczna liczba godzin? Odpowiedź powinna być oczywista.

Czasami względy wydajnościowe powodują, że suma jest obliczana w czasie akceptacji kart pracy, a nie w czasie drukowania raportu. Jest to do przyjęcia pod warunkiem, że nazwy funkcji będą to odzwierciedlać. W takim przypadku w module kart czasu pracy powinna znajdować się funkcja `computeRunningTotalOfHours`.

G18. Niewłaściwe metody statyczne

Przykładem dobrej metody statycznej jest `Math.max(double a, double b)`. Nie działa ona na jednym obiekcie, a w rzeczywistości niewłaściwe byłoby korzystanie z `new Math().max(a, b)` czy też `a.max(b)`. Wszystkie dane wykorzystywane przez `max` pochodzą z argumentów, a nie z „zawierającego" ją obiektu. Co więcej, niemal *nie ma szansy*, abyśmy potrzebowali polimorficznej metody `Math.max`.

Czasami jednak piszemy metody statyczne, które nie powinny być statyczne. Oto przykład:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Wydaje się, że jest to sensowna funkcja statyczna. Nie operuje ona na żadnym określonym obiekcie i pobiera wszystkie dane z argumentów. Jednak istnieje prawdopodobieństwo, że będziemy potrzebować, aby funkcja ta była polimorficzna. Możemy chcieć zaimplementować kilka różnych algorytmów obliczania płacy godzinowej, na przykład `OvertimeHourlyPayCalculator`, czasami `StraightTimeHourlyPayCalculator`. Dlatego w tym przypadku funkcja nie powinna być statyczna. Powinna być niestatyczną metodą składową klasy `Employee`.

Powinniśmy preferować korzystanie z metod niestatycznych zamiast statycznych. Jeżeli mamy wątpliwości, nie tworzymy funkcji statycznej. Jeżeli faktycznie chcemy, aby funkcja była statyczna, musimy się upewnić, że nie powinna ona działać polimorficznie.

G19. Użycie opisowych zmiennych

Kent Beck pisał o tym w swojej wspaniałej książce *Smalltalk Best Practice Patterns*[8] oraz później, w równie dobrej pracy *Implementation Patterns*[9]. Jednym z najefektywniejszych sposobów zapewnienia czytelności programu jest podzielenie obliczeń na kroki pośrednie, których wyniki są przechowywane w zmiennych o znaczących nazwach.

Przykładem może być fragment kodu z FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

Dzięki użyciu zmiennych opisowych od razu wiadomo, że pierwsza znaleziona grupa jest *kluczem*, a druga *wartością*.

Trudno zrobić to lepiej. Im więcej zmiennych opisowych, tym lepiej. Trudno uwierzyć, jak szybko nieczytelny moduł staje się przejrzysty, gdy tylko podzielimy obliczenia na dobrze nazwane wartości pośrednie.

G20. Nazwy funkcji powinny informować o tym, co realizują

Spójrzmy na ten wiersz kodu:

```
Date newDate = date.add(5);
```

Czy możemy oczekiwać, że spowoduje to dodanie pięciu dni do daty? Czy też tygodni lub godzin? Czy obiekt `date` jest zmieniany, czy tylko funkcja zwraca nowy obiekt `Date` bez zmiany starego? *Nie jesteśmy w stanie stwierdzić na podstawie wywołania, co robi funkcja.*

Jeżeli funkcja dodaje pięć dni do daty i zmienia ją, to powinna być nazwana addDaysTo lub `increaseByDays`. Jeżeli jednak funkcja zwraca nową datę późniejszą o pięć dni i nie zmienia bazowego obiektu daty, powinna być nazwana `daysLater` lub `daysSince`.

Jeżeli musimy spojrzeć do implementacji (lub dokumentacji) funkcji w celu sprawdzenia, co ona robi, to powinniśmy znaleźć jej lepszą nazwę lub zmienić działanie tak, aby mogło być realizowane przez funkcje o lepszych nazwach.

---

[8] [Beck97], s. 108.

[9] [Beck07].

G21. Zrozumienie algorytmu

Dzięki temu, że ludzie nie poświęcają czasu na zrozumienie algorytmu, powstało wiele śmiesznego kodu. Próbują oni uzyskać jakiekolwiek wyniki przez dodawanie odpowiedniej liczby instrukcji `if` oraz znaczników, zamiast zatrzymać się i sprawdzić, co naprawdę powinni zrobić.

Programowanie często jest eksploracją. *Uważamy*, że znamy odpowiedni algorytm czynności, ale następnie męczymy się z nim, ugniatając go i szturchając, do momentu, aż będzie on „działał". Skąd jednak wiemy, że „działa"? Ponieważ realizuje wszystkie testy, o których pomyśleliśmy.

Nie ma nic złego w takim podejściu. W rzeczywistości często jest to jedyny sposób na otrzymanie potrzebnej funkcji. Jednak nie można pozostawiać cudzysłowów wokół „działa".

Zanim uznamy, że funkcja jest zakończona, upewnijmy się, że *rozumiemy*, jak ona działa. Nie wystarczy, że przechodzi wszystkie testy. Musimy *wiedzieć*[10], że rozwiązanie jest prawidłowe.

Najczęściej najlepszym sposobem na uzyskanie tej wiedzy i zrozumienie algorytmu jest przebudowa funkcji do postaci tak jasnej i ekspresywnej, że jest *oczywiste*, jak ona działa.

G22. Zamiana zależności logicznych na fizyczne

Jeżeli jeden moduł zależy od drugiego, to ta zależność powinna być fizyczna, a nie tylko logiczna. Moduł zależny nie powinien wykonywać założeń (inaczej mówiąc, zależności logicznych) na temat modułu, od którego zależy. Zamiast tego należy jawnie odpytać ten moduł o informacje, na których nam zależy.

Załóżmy, że piszemy funkcję generującą tekstowy raport godzin przepracowanych przez pracowników. Klasa o nazwie `HourlyReporter` zbiera wszystkie dane, a następnie przekazuje je do `Hourly ↪ReportFormatter` w celu ich wyświetlenia (patrz listing 17.1).

LISTING 17.1. HourlyReporter.java

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee> employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
```

[10] Istnieje różnica pomiędzy znajomością działania kodu a pewnością, że algorytm realizuje wymagane działanie. Brak pewności co do algorytmu zdarza się, i często jest po prostu faktem. Brak pewności co do działania własnego kodu jest po prostu lenistwem.

```
        printAndClearItemList();
      }
      if (page.size() > 0)
        printAndClearItemList();
    }

    private void printAndClearItemList() {
      formatter.format(page);
      page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
      LineItem item = new LineItem();
      item.name = e.getName();
      item.hours = e.getTenthsWorked() / 10;
      item.tenths = e.getTenthsWorked() % 10;
      page.add(item);
    }

    public class LineItem {
      public String name;
      public int hours;
      public int tenths;
    }
  }
```

Kod ten zawiera zależności logiczne, które nie zostały fizycznie ujawnione. Czy możesz je znaleźć? Jest to stała `PAGE_SIZE`. Dlaczego klasa `HourlyReporter` zna wielkość strony? Wielkość ta powinna być odpowiedzialnością klasy `HourlyReportFormatter`.

Fakt deklaracji stałej `PAGE_SIZE` w `HourlyReporter` jest źle rozmieszczoną odpowiedzialnością [G17], co powoduje, że `HourlyReporter` zakłada, iż wie, jaka powinna być wielkość strony. Takie założenie jest zależnością logiczną. Klasa `HourlyReporter` zależy od tego, że `HourlyReportFormatter` może obsługiwać strony o długości 55 wierszy. Jeżeli pewna implementacja `HourlyReportFormatter` nie będzie obsługiwała takiej wielkości, wystąpi błąd.

Można ujawnić tę zależność przez utworzenie w klasie `HourlyReportFormatter` nowej metody o nazwie `getMaxPageSize()`. Klasa `HourlyReporter` powinna wywołać tę funkcję, zamiast korzystać ze stałej `PAGE_SIZE`.

G23. Zastosowanie polimorfizmu zamiast instrukcji if-else lub switch-case

Wydaje się to dziwną sugestią po przedstawieniu zagadnienia w rozdziale 6. W końcu uznałem tam, że instrukcje `switch` są lepsze w tych częściach systemu, w których dodawanie nowych funkcji jest bardziej prawdopodobne niż dodawanie nowych typów.

Po pierwsze, większość programistów używa instrukcji `switch`, ponieważ jest to rozwiązanie oczywiste, a nie dlatego, że jest to rozwiązanie właściwe w danej sytuacji. Poruszamy ten temat, aby przypomnieć, że przed zastosowaniem instrukcji `switch` należy rozważyć użycie polimorfizmu.

Po drugie, przypadki, w których funkcje są bardziej ulotne od typów, są względnie rzadkie. Dlatego każda instrukcja `switch` powinna być dla nas podejrzana.

Osobiście korzystam z następującej zasady „jeden `switch`": *W danej deklaracji typu może znaleźć się najwyżej jedna instrukcja `switch`. W takim przypadku instrukcja `switch` powinna tworzyć obiekty polimorficzne, które zastępują instrukcje `switch` w pozostałych częściach systemu*.

G24. Wykorzystanie standardowych konwencji

Każdy zespół powinien korzystać ze standardów kodowania bazujących na wspólnych normach przemysłowych. Standard kodowania powinien definiować takie zagadnienia, jak miejsce deklaracji zmiennych instancyjnych, sposób nazewnictwa klas, metod i zmiennych, miejsce umieszczenia klamer i tak dalej. Zespół nie powinien potrzebować dokumentu opisującego te konwencje, ponieważ przykłady są zawarte w kodzie.

Każdy członek zespołu powinien stosować się do tych konwencji. Oznacza to, że każdy członek zespołu powinien być na tyle dojrzały, aby zgodzić się, że nie ma znaczenia, gdzie umieszcza się klamry, dopóki wszyscy zgadzają się na jedno miejsce.

Jeżeli ktoś chciałby znać używane przeze mnie konwencje, może je zobaczyć w przebudowanym kodzie zamieszczonym w listingach od B.7 do B.14 w dodatku B.

G25. Zamiana magicznych liczb na stałe nazwane

Jest to prawdopodobnie jedna z najstarszych zasad programowania. Pamiętam ją już z podręczników Cobola, Fortrana i PL/1 z lat sześćdziesiątych ubiegłego wieku. Zwykle złym pomysłem jest korzystanie z surowych liczb w kodzie. Powinniśmy je ukrywać w dobrze nazwanych stałych.

Na przykład liczba 86 400 powinna być ukryta w stałej `SECONDS_PER_DAY`. Jeżeli wyświetlamy 55 wierszy na stronę, to liczba 55 powinna być ukryta w stałej `LINES_PER_PAGE`.

Niektóre stałe są tak proste do rozpoznania, że nie zawsze wymagają stałej nazwanej, o ile są połączone z bardzo dobrze opisującym się kodem. Na przykład:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Czy faktycznie w powyższych przykładach potrzebujemy stałych `FEET_PER_MILE`, `WORK_HOURS_PER_DAY` oraz `TWO`? Oczywiście, ostatni przypadek jest absurdalny. Istnieją jednak wyrażenia, w których stałe są po prostu lepsze niż surowe liczby. Można mieć wątpliwości co do stałej `WORK_HOURS_PER_DAY`, ponieważ prawo i konwencje mogą się zmieniać. Z drugiej strony, wyrażenie to czyta się tak dobrze z cyfrą, że byłbym ostrożny przed dodawaniem do niego 17 znaków. W przypadku `FEET_PER_MILE` liczba 5280 jest tak dobrze znana i unikatowa, że czytelnik rozpozna ją, nawet gdyby znajdowała się sama na stronie, bez żadnego kontekstu.

Stałe, takie jak 3,141592653589793, są również bardzo dobrze znane i łatwo rozpoznawalne. Jednak szansa popełnienia błędu jest zbyt duża, aby pozostawić ją w surowej postaci. Za każdym razem, gdy ktoś zobaczy 3,1415927535890793, będzie wiedział, że jest to *pi*, i nie będzie jej sprawdzał (czy zauważyłeś błąd w jednej cyfrze?). Nie chcemy również, aby używane były wartości 3,14, 3,14159, 3,142 i tak dalej. Dlatego dobrze, że mamy już zdefiniowaną stałą `Math.PI`.

Termin „magiczna liczba" nie odnosi się wyłącznie do liczb. Odnosi się do dowolnej wartości, która nie jest rozpoznawalna na pierwszy rzut oka. Na przykład:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

W tej asercji mamy dwie magiczne liczby. Pierwszą jest 7777, ponieważ jej znaczenie nie jest oczywiste. Drugą magiczną liczbą jest `"John Doe"`, ponieważ i w tym przypadku intencje nie są jasne.

Wygląda to tak, jakby `"John Doe"` był nazwą pracownika numer 7777 w testowej bazie danych, utworzonej przez nasz zespół. Każdy w zespole wie, że po podłączeniu do tej bazy będzie miał dostęp do danych kilku pracowników mających znane wartości i atrybuty. Okazuje się więc, że `"John Doe"` reprezentuje jedynego pracownika opłacanego na godziny w tej testowej bazie danych. Dlatego test ten powinien wyglądać następująco:

```
assertEquals(
  HOURLY_EMPLOYEE_ID,
  Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26. Precyzja

Oczekiwanie, że pierwsze dopasowanie będzie *jedynym* wynikiem zapytania, jest co najmniej naiwne. Użycie liczb zmiennoprzecinkowych do reprezentowania walut jest niemal przestępstwem. Unikanie blokad i zarządzania transakcjami, gdyż nie uważamy, że równoległa aktualizacja jest prawdopodobna, jest lenistwem. Deklarowanie zmiennej jako `ArrayList`, gdy wystarczy `List`, jest nadmiernym ograniczeniem. Domyślne oznaczanie wszystkich zmiennych jako `protected` jest niewystarczającym ograniczeniem.

Podejmując decyzje dotyczące kodu, należy to robić *precyzyjnie*. Trzeba wiedzieć, dlaczego są one podjęte i jak obsłużyć wszystkie wyjątki. Nie należy być leniwym, jeżeli chodzi o precyzję decyzji. Jeżeli zdecydujemy, że funkcja może zwracać `null`, należy upewnić się, że wartość `null` jest kontrolowana. Jeżeli zadajemy zapytanie dotyczące elementu, który powinien być jedyny w bazie danych, nasz kod powinien sprawdzić, czy nie ma innych. Jeżeli potrzebujemy obsługi walut, należy użyć liczb całkowitych[11] i prawidłowo obsługiwać zaokrąglanie. Jeżeli istnieje możliwość równoległej aktualizacji, należy zaimplementować mechanizm blokowania.

Niejednoznaczności w kodzie są wynikiem niezgodności lub lenistwa. Bez względu na ich przyczynę, powinny zostać wyeliminowane.

[11] Lub lepiej — zastosować klasę `Money` wykorzystującą liczby całkowite.

G27. Struktura przed konwencją

Decyzje projektowe należy wymuszać przez zastosowanie struktury przed konwencją. Konwencje nazewnictwa są dobre, ale są gorsze od struktur wymuszających zgodność. Na przykład instrukcje `switch-case` z ładnie nazwanymi wyliczeniami są gorsze od klas bazowych z metodami abstrakcyjnymi. Nikt nie jest zmuszany do implementowania instrukcji `switch-case` zawsze w taki sam sposób, natomiast klasy bazowe mogą wymusić, aby klasy konkretne miały zaimplementowane wszystkie metody abstrakcyjne.

G28. Hermetyzacja warunków

Wyrażenia logiczne są wystarczająco trudne do zrozumienia, bez konieczności oglądania kontekstu instrukcji `if` lub `while`. Warto wyodrębniać funkcję, która wyjaśnia przeznaczenie warunku.

Na przykład:

```
if (shouldBeDeleted(timer))
```

jest lepsze niż

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29. Unikanie warunków negatywnych

Warunki negatywne są nieco trudniejsze do zrozumienia niż pozytywne. Dlatego, jeżeli jest to możliwe, wyrażenia powinny być formułowane jako warunki pozytywne. Na przykład:

```
if (buffer.shouldCompact())
```

jest lepsze niż

```
if (!buffer.shouldNotCompact())
```

G30. Funkcje powinny wykonywać jedną operację

Często kuszące jest tworzenie funkcji, które mają wiele sekcji wykonujących serię działań. Funkcje tego rodzaju realizują więcej niż *jedną operację* i powinny zostać zamienione na mniejsze, z których każda wykonuje *jedną operację*.

Na przykład:

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

Ten fragment kodu wykonuje trzy operacje. Przegląda dane pracowników, sprawdza, czy kolejnemu pracownikowi należy zapłacić, a następnie realizuje wypłatę. Lepiej zapisać go tak:

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

Teraz każda z tych funkcji wykonuje jedną operację (patrz „Wykonuj jedną czynność" w rozdziale 3.).

G31. Ukryte sprzężenia czasowe

Sprzężenia czasowe są często potrzebne, ale nie powinny ukrywać zależności. Należy tak uporządkować argumenty funkcji, aby kolejność ich wywoływania była oczywista. Weźmy pod uwagę następujący kod:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    saturateGradient();
    reticulateSplines();
    diveForMoog(reason);
  }
  ...
}
```

Kolejność tych trzech funkcji jest ważna. Należy nasycić gradient przed retykulacją splajnów i tylko po tej operacji można wykonać ostatnią funkcję. Niestety, kod ten nie wymusza tego sprzężenia czasowego. Inny programista może wywołać `reticulateSplines` przed `saturateGradient`, doprowadzając do wyjątku `UnsaturatedGradientException`. Lepszym rozwiązaniem jest następująca konstrukcja:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
  }
  ...
}
```

Ujawnia to sprzężenie czasowe przez utworzenie zestawu kubełków. Każda funkcja wytwarza wynik potrzebny następnej funkcji, więc nie ma sensownego sposobu na wywołanie ich w nieprawidłowej kolejności.

Można narzekać, że zwiększa to złożoność funkcji, i trzeba się z tym zgodzić. Jednak dodatkowa złożoność składniowa ujawnia prawdziwe sprzężenie czasowe w tej sytuacji.

Należy zwrócić uwagę, że pozostawiłem zmienne instancyjne. Zakładam, że będą one potrzebne metodom prywatnym w tej klasie. Nawet pomimo tego chcemy, aby sprzężenia czasowe były ujawnione.

G32. Unikanie dowolnych działań

Struktura kodu nie powinna być przypadkowa. Jeżeli struktura wygląda na dowolną, inne osoby czują się zobowiązane do jej zmiany. Jeżeli struktura jest spójna w całym systemie, inne osoby będą jej używać w celu zachowania konwencji. Na przykład ostatnio łączyłem zmiany w FitNesse i odkryłem, że jeden z programistów dodał taki kod:

```
public class AliasLinkWidget extends ParentWidget
{
  public static class VariableExpandingWidgetRoot {
  ...
  ...
}
```

Problemem jest to, że `VariableExpandingWidgetRoot` nie musi być w zakresie `AliasLinkWidget`. Co więcej, inne niezwiązane klasy korzystały z `AliasLinkWidget.VariableExpandingWidgetRoot`. Klasy te nie musiały nic wiedzieć na temat `AliasLinkWidget`.

Prawdopodobnie programista wrzucił `VariableExpandingWidgetRoot` do `AliasWidget` dla wygody albo myślał, że faktycznie będzie potrzebował operacji wewnątrz `AliasWidget`. Niezależnie od powodu, wynik wygląda na dosyć dowolny. Klasy publiczne, które nie są narzędziami innej klasy, nie powinny być zagłębione w innej klasie. Zgodnie z konwencją, należy zadeklarować je jako publiczne na poziomie pakietu.

G33. Hermetyzacja warunków granicznych

Warunki graniczne są trudne do obsługi. Należy umieścić ich przetwarzanie w jednym miejscu. Nie należy dopuszczać, aby rozlały się po całym kodzie. Nie chcemy stada `+1` i `-1` wyglądających zza każdego rogu. Przykładem może być fragment kodu z FIT:

```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

Można zauważyć, że `level+1` występuje dwa razy. Jest to warunek graniczny, który powinien być hermetyzowany wewnątrz zmiennej o nazwie podobnej do `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

G34. Funkcje powinny zagłębiać się na jeden poziom abstrakcji

Instrukcje wewnątrz funkcji powinny być pisane na tym samym poziomie abstrakcji, który powinien być o jeden poziom niższy od operacji opisanych w nazwie funkcji. Może być to najtrudniejsza do interpretacji i stosowania z wszystkich opisanych tu heurystyk. Choć zagadnienie jest dosyć proste, ludzie są po prostu zbyt dobrzy w płynnym mieszaniu poziomów abstrakcji. Jako przykład weźmy następujący kod z FitNesse:

```
public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if(size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");
  return html.toString();
}
```

Krótka analiza pozwala stwierdzić, co się tu dzieje. Funkcja ta tworzy znacznik HTML rysujący poziomą linię na stronie. Wysokość tej linii jest zdefiniowana w zmiennej `size`.

Teraz spójrzmy na niego jeszcze raz. Metoda ta miesza ze sobą co najmniej dwa poziomy abstrakcji. Pierwszym jest informacja, że linia pozioma posiada wielkość. Drugim jest sama składnia znacznika `HR`. Kod ten pochodzi z modułu `HruleWidget` z FitNesse. Moduł ten wykrywa wiersz z czterema lub więcej minusami i konwertuje je na odpowiedni znacznik `HR`. Im więcej minusów, tym większy rozmiar.

Poniżej zamieszczony jest kod po przebudowie. Należy zauważyć, że zmieniłem nazwę pola `size`, aby odzwierciedlała jego prawdziwe przeznaczenie. Przechowuje ono liczbę dodatkowych minusów.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}

private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

Zmiana ta pozwoliła na eleganckie rozdzielenie dwóch poziomów abstrakcji. Funkcja `render` tworzy dla nas znacznik `HR`, bez konieczności znajomości składni HTML tego znacznika. Moduł `HtmlTag` zajmuje się wszystkimi problemami składniowymi.

Wprowadzając tę zmianę, znalazłem subtelny błąd. Oryginalny kod nie umieszczał zamykającego ukośnika znacznika HR, tak jak definiuje to standard XHTML (inaczej mówiąc, generował <hr> zamiast <hr/>). Moduł HtmlTag został już dawno tak zmieniony, aby był zgodny z XHTML-em.

Rozdzielenie poziomów abstrakcji jest jednym z najważniejszych zadań przebudowy, przy tym najtrudniejszym do wykonania. Jako przykład weźmy poniższy kod. Była to moja pierwsza próba rozdzielenia poziomów abstrakcji w metodzie HruleWidget.render.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (size > 0) {
    hr.addAttribute("size", ""+(size+1));
  }
  return hr.html();
}
```

Moim celem było zapewnienie odpowiedniej separacji i umożliwienie wykonywania testów. Łatwo zrealizowałem te cele, ale wynikiem była funkcja, która *nadal* mieszała poziomy abstrakcji. W tym przypadku wymieszanymi poziomami były: tworzenie znacznika HR oraz interpretacja i formatowanie zmiennej size. W tym przypadku okazało się, że gdy podzielimy funkcję wzdłuż poziomów abstrakcji, często wykrywamy następne, ukryte przez poprzednią strukturę.

G35. Przechowywanie danych konfigurowalnych na wysokim poziomie

Jeżeli mamy stałe, takie jak wartości domyślne lub konfiguracyjne, które są oczekiwane na wysokim poziomie abstrakcji, to nie należy ukrywać ich w funkcjach niskiego poziomu. Należy ujawnić je jako argumenty, z użyciem których funkcje niskiego poziomu są wywoływane z funkcji wysokiego poziomu. Weźmy pod uwagę następujący kod z FitNesse:

```
public static void main(String[] args) throws Exception
{
  Arguments arguments = parseCommandLine(args);
  ...
}

public class Arguments
{
  public static final String DEFAULT_PATH = ".";
  public static final String DEFAULT_ROOT = "FitNesseRoot";
  public static final int DEFAULT_PORT = 80;
  public static final int DEFAULT_VERSION_DAYS = 14;
  ...
}
```

Argumenty wiersza poleceń są analizowane w pierwszym wykonywalnym wierszu FitNesse. Wartości domyślne tych argumentów są specyfikowane w klasie Arguments. Nie musimy zaglądać do niskich poziomów systemu za pomocą takich instrukcji:

```
if (arguments.port == 0) // Używamy domyślnie 80.
```

Stałe konfiguracji znajdują się na bardzo wysokim poziomie i łatwo je zmienić. Są one przekazywane w dół, do pozostałych części aplikacji. Niższe poziomy aplikacji nie zawierają wartości tych stałych.

G36. Unikanie nawigacji przechodnich

Zazwyczaj nie chcemy, aby jeden moduł znał swoich współpracowników. Mówiąc dokładniej, jeżeli `A` współdziała z `B`, a `B` współpracuje z `C`, to nie chcemy, aby moduły korzystające z `A` wiedziały coś o `C` (na przykład nie chcemy instrukcji `a.getB().getC().doSomething();`).

Jest to czasami nazywane prawem Demeter. Pragmatyczni programiści nazywają to „pisaniem wstydliwego kodu”[12]. W obu przypadkach sprowadza się to do upewnienia się, że moduły znają tylko swoich bezpośrednich współpracowników i nie znają mapy nawigacji w całym systemie.

Jeżeli wiele modułów korzysta z instrukcji takich jak `a.getB().getC()`, to trudno zmienić projekt i architekturę i wstawić `Q` pomiędzy `B` a `C`. Konieczne będzie wyszukanie wszystkich wystąpień `a.getB().getC()` i ich zamiana na `a.getB().getQ().getC()`. W ten sposób architektura staje się sztywna. Zbyt wiele modułów wie zbyt dużo o architekturze.

Zamiast tego oczekujemy, że pośredni współpracownicy będą oferowali wszystkie potrzebne usługi. Nie powinniśmy przeglądać grafu obiektów w systemie, polując na metodę do wywołania. Zamiast tego powinniśmy być w stanie skorzystać z:

```
myCollaborator.doSomething().
```

Java

J1. Unikanie długich list importu przez użycie znaków wieloznacznych

Jeżeli używamy dwóch lub więcej klas z pakietu, to warto zaimportować go w całości przez użycie:

```
import package.*;
```

Długie listy importu są nużące dla czytelnika. Nie chcemy zaśmiecać początku modułu ponad 80 wierszami importów. Chcemy raczej, aby importy były zwięzłymi instrukcjami opisującymi pakiety, z którymi współpracujemy.

Specyficzne importy są trwałymi zależnościami, natomiast importy ze znakami wieloznacznymi — nie. Jeżeli zaimportujemy specyficzną klasę, to *musi* ona istnieć. Jeżeli jednak zaimportujemy pakiet z użyciem znaków wieloznacznych, nie musi istnieć żadna określona klasa. Instrukcja importu po prostu dodaje pakiet do ścieżki przeszukiwania, używanej przy polowaniu na nazwy. Dlatego za pomocą importu nie są tworzone konkretne zależności, dzięki czemu moduły są mniej sprzężone.

---

[12] [PRAG], s. 138.

Istnieją przypadki, w których długie listy importów mogą być przydatne. Na przykład, jeżeli korzystamy z zastanego kodu i musimy dowiedzieć się, których klas potrzebujemy w celu zbudowania imitacji i zrębów, możemy przeglądać listę specyficznych importów, szukając nazw kwalifikowanych wszystkich tych klas, a następnie zastąpić je zrębami. Jednak takie zastosowanie specyficznych importów jest bardzo rzadkie. Dodatkowo większość nowoczesnych środowisk IDE pozwala na zamianę importów ze znakami wieloznacznymi na listę specyficznych importów po wykonaniu jednego polecenia. Dlatego nawet w zastanym kodzie lepiej korzystać z importów wieloznacznych.

Importy wieloznaczne mogą czasami powodować konflikty nazw i niejednoznaczności. Dwie klasy o tej samej nazwie, ale znajdujące się w różnych pakietach, będą wymagały specyficznego importu lub co najmniej kwalifikowania w czasie użycia. Może to być niuans, ale jest tak rzadki, że najczęściej importowanie z użyciem znaków wieloznacznych jest lepsze od specyficznych importów.

J2. Nie dziedziczymy stałych

Widziałem to kilka razy i zawsze wywołuje to grymas na mojej twarzy. Programista umieszcza stałe w interfejsie, a następnie uzyskuje dostęp do tych stałych przez dziedziczenie interfejsu. Przyjrzyjmy się następującemu przykładowi kodu:

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

Skąd pochodzą stałe `TENTHS_PER_WEEK` oraz `OVERTIME_RATE`? Mogą pochodzić z klasy `Employee`, więc rzućmy na nią okiem:

```
public abstract class Employee implements PayrollConstants {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

Nie, nie ma ich tu. No to gdzie są? Spójrzmy dokładniej na klasę `Employee`. Implementuje ona `PayrollConstants`.

```
public interface PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  public static final double OVERTIME_RATE = 1.5;
}
```

Jest to fatalna praktyka! Stałe są ukryte na szczycie hierarchii dziedziczenia. Okropność! Nie należy używać dziedziczenia jako sposobu oszukania zasad zakresu zdefiniowanych w języku. Zamiast tego należy korzystać z importu statycznego.

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

J3. Stałe kontra typy wyliczeniowe

Teraz, gdy do języka zostało dodane słowo kluczowe `enum` (Java 5), warto z niego korzystać! Nie należy trzymać się starych sztuczek `public static final int`. Znaczenie wartości `int` może zostać utracone. Znaczenie typu wyliczeniowego nie może, ponieważ należy do nazwanego wyliczenia.

Co więcej, warto uważnie przestudiować składnię `enum`. Mogą one posiadać metody i pola. Powoduje to, że stają się bardzo potężnym narzędziem i pozwalają na uzyskanie znacznie większej ekspresywności i elastyczności niż liczby `int`. Przeanalizujmy odmianę wcześniejszego kodu obliczania płac:

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  HourlyPayGrade grade;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
...
}

public enum HourlyPayGrade {
  APPRENTICE {
    public double rate() {
      return 1.0;
    }
  },
  LEUTENANT_JOURNEYMAN {
    public double rate() {
      return 1.2;
    }
  },
  JOURNEYMAN {
    public double rate() {
      return 1.5;
    }
  },
  MASTER {
    public double rate() {
```

```
        return 2.0;
      }
    };

    public abstract double rate();
  }
```

Nazwy

N1. Wybór opisowych nazw

Nie należy zbyt szybko wybierać nazwy. Należy upewnić się, że jest opisowa. Trzeba pamiętać, że znaczenie ma tendencję do dryfowania wraz z rozwojem oprogramowania, więc częste sprawdzanie dokładności nazw jest wskazane.

To nie jest tylko rekomendacja „dobrego samopoczucia". W 90 procentach nazwy w kodzie decydują o jego czytelności. Nazwy są zbyt ważne, aby je traktować beztrosko.

Weźmy pod uwagę poniższy kod. Co on robi? Jeżeli pokażę go z użytymi odpowiednimi nazwami, będzie dla nas zupełnie sensowny, ale w tej postaci jest zlepkiem symboli i magicznych liczb.

```
public int x() {
  int q = 0;
  int z = 0;
  for (int kk = 0; kk < 10; kk++) {
    if (l[z] == 10)
    {
      q += 10 + (l[z + 1] + l[z + 2]);
      z += 1;
    }
    else if (l[z] + l[z + 1] == 10)
    {
      q += 10 + l[z + 2];
      z += 2;
    } else {
      q += l[z] + l[z + 1];
      z += 2;
    }
  }
  return q;
}
```

Poniżej mamy ten sam kod zapisany we właściwy sposób. Ten fragment jest w rzeczywistości mniej kompletny niż pokazany powyżej. Niemal natychmiast można wywnioskować, co on ma robić, i z dużym prawdopodobieństwem moglibyśmy od razu napisać brakujące funkcje, bazując tylko na wywnioskowanym znaczeniu. Magiczne liczby przestały być magiczne, a struktura algorytmu jest zachęcająco opisowa.

```
public int score() {
  int score = 0;
  int frame = 0;
  for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
      score += 10 + nextTwoBallsForStrike(frame);
      frame += 1;
```

```
    } else if (isSpare(frame)) {
      score += 10 + nextBallForSpare(frame);
      frame += 2;
    } else {
      score += twoBallsInFrame(frame);
      frame += 2;
    }
  }
  return score;
}
```

Siłą dobrze dobranych nazw jest uzupełnianie struktury kodu przy użyciu opisów. Pozwala to na przystosowanie oczekiwań czytelnika do tego, co realizują inne funkcje w module. Czy możemy wywnioskować implementację `isStrike()`, spoglądając na powyższy kod? Gdy będziemy czytać metodę `isStrike`, będzie robiła „mniej więcej to, czego oczekiwaliśmy”[13].

```
private boolean isStrike(int frame) {
  return rolls[frame] == 10;
}
```

N2. Wybór nazw na odpowiednich poziomach abstrakcji

Nie należy wybierać nazw informujących o implementacji, ale raczej nazwy odzwierciedlające poziom abstrakcji, na którym operuje klasa lub funkcja. Jest to trudne do realizacji. Przypomnijmy, że ludzie są zbyt dobrzy w mieszaniu poziomów abstrakcji. Za każdym razem, gdy przejrzysz swój kod, najprawdopodobniej znajdziesz kilka zmiennych, których nazwy są na zbyt niskim poziomie. Po znalezieniu takich nazw warto wykorzystać okazję i zmienić je. Proces tworzenia czytelnego kodu wymaga chęci do stałej poprawy. Weźmy jako przykład poniższy interfejs `Modem`:

```
public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedPhoneNumber();
}
```

Na pierwszy rzut oka wygląda świetnie. Wszystkie funkcje wydają się odpowiednie. Faktycznie, dla wielu zastosowań są takie. Weźmy pod uwagę aplikację, w której modemy nie są łączone z użyciem wybierania. Zamiast tego są połączone na stałe (tak jak modemy kablowe używane obecnie do dostarczania internetu w wielu domach). Być może niektóre są łączone przez wysyłanie numeru portu do przełącznika, z użyciem połączenia USB. Jasne jest, że notacja numerów telefonów znajduje się na niewłaściwym poziomie abstrakcji. Lepszą strategią nazewniczą dla tego scenariusza może być:

```
public interface Modem {
  boolean connect(String connectionLocator);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedLocator();
}
```

[13] Patrz cytat wypowiedzi Warda Cunninghama z rozdziału 1.

Teraz nazwy nie zawierają żadnych odwołań do numerów telefonicznych. Numery telefoniczne mogą być nadal używane, można też zastosować inną strategię połączenia.

N3. Korzystanie ze standardowej nomenklatury tam, gdzie jest to możliwe

Nazwy są łatwiejsze do zrozumienia, jeżeli bazują na istniejących konwencjach lub zastosowaniach. Na przykład, jeżeli korzystamy z wzorca Dekorator, to powinniśmy użyć słowa `Decorator` w nazwach klas dekorujących. Nazwą klasy dekorującej `Modem` i zapewniającej automatyczne rozłączanie na końcu sesji może być `AutoHangupModemDecorator`.

Wzorce są po prostu jednym z rodzajów standardów. W języku Java funkcje konwertujące obiekty na postać znakową są często nazywane `toString`. Lepiej zachować zgodność z tego typu konwencjami, niż wymyślać własne.

Zespoły często wymyślają własne standardy nazewnictwa dla określonych projektów. Eric Evans nazywa to *wszechobecnym językiem* projektu[14]. W naszym kodzie powinniśmy intensywnie używać terminów z tego języka. Mówiąc krótko, im więcej używamy nazw przeładowanych znaczeniami odnoszącymi się do naszego projektu, tym łatwiej będzie czytelnikowi zrozumieć, do czego służy dany kod.

N4. Jednoznaczne nazwy

Należy wybierać nazwy, które nie pozostawiają niejednoznaczności co do działania funkcji lub zmiennej. Spójrzmy na fragment kodu z FitNesse:

```
private String doRename() throws Exception
{
  if(refactorReferences)
    renameReferences();
  renamePage();
  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

Nazwa tej funkcji nie mówi nam, co funkcja realizuje, poza ogólnym i niejasnym pojęciem. Odczucie to jest jeszcze powiększane przez fakt, że wewnątrz funkcji `doRename` znajduje się wywołanie funkcji `renamePage`! Co te nazwy mówią nam o różnicach pomiędzy tymi dwoma funkcjami? Nic.

Lepszą nazwą dla tej funkcji byłaby `renamePageAndOptionallyAllReferences`. Może się ona wydawać długa (bo taka faktycznie jest), ale funkcja jest wywoływana z jednego miejsca w module, więc wartość informacyjna tej nazwy przewyższa problemy wynikające z jej długości.

[14] [DDD].

N5. Użycie długich nazw dla długich zakresów

Długość nazwy powinna być związana z długością zakresu. Można używać bardzo krótkich nazw dla niewielkich zakresów, ale dla dużych zakresów powinny być używane długie nazwy.

Nazwy zmiennych takie jak `i` oraz `j` są świetne, o ile ich zakres ma długość najwyżej pięciu wierszy. Oto fragment kodu starej gry w kręgle:

```
private void rollMany(int n, int pins)
{
  for (int i=0; i<n; i++)
    g.roll(pins);
}
```

Jest to zupełnie jasne i jeżeli zmienna `i` zostałaby zastąpiona nieco irytującą, jak `rollCount`, kod byłby trudniejszy do zrozumienia. Z drugiej strony, zmienne i funkcje o krótkich nazwach tracą swoje znaczenie na dłuższych dystansach. Dlatego im szerszy zakres nazwy, tym powinna być ona dłuższa i precyzyjniejsza.

N6. Unikanie kodowania

W nazwach nie powinny być kodowane informacje o typie i zakresie. Przedrostki, takie jak `m_` czy `f`, są w dzisiejszych środowiskach bezużyteczne. Również kodowanie projektu lub (i) podsystemu, takiego jak `vis_` (dla systemu przetwarzania wizualnego), jest rozpraszające i nadmiarowe. Stosowane obecnie środowiska zapewniają wszystkie te informacje bez konieczności dołączania ich do nazw. Należy zachować nazwy wolne od skażenia notacją węgierską.

N7. Nazwy powinny opisywać efekty uboczne

Nazwy powinny opisywać wszystko, co wykonuje dana funkcja, zmienna lub klasa. Nie należy ukrywać efektów ubocznych w nazwie. Nie należy używać prostego słowa do opisania funkcji, która wykonuje więcej niż tylko tę prostą akcję. Przykładem może być następujący kod z TestNG:

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

Funkcja ta realizuje więcej niż tylko pobieranie „oos”; ona tworzy „oos”, jeżeli nie był wcześniej utworzony. Dlatego lepszą nazwą może być `createOrReturnOos`.

Testy

T1. Niewystarczające testy

Ile testów powinno znajdować się w zestawie testowym? Niestety, wielu programistów stosuje miarę „to powinno wystarczyć". Zestaw testów powinien sprawdzać wszystko, co może zawieść. Testy są niewystarczające dopóty, dopóki będą istnieć warunki nieskontrolowane przez testy lub niesprawdzone obliczenia.

T2. Użycie narzędzi kontroli pokrycia

W naszej strategii testowania konieczne jest wykorzystanie narzędzi raportujących pokrycie kodu. Ułatwiają one znalezienie modułów, klas i funkcji, które są niewystarczająco testowane. Większość środowisk IDE posiada wizualną prezentację pokrycia, oznaczając wiersze pokryte testami kolorem zielonym, a niepokryte kolorem czerwonym. Pozwala to szybko i łatwo znaleźć instrukcje `if` oraz `catch`, których treść nie jest sprawdzana.

T3. Nie pomijaj prostych testów

Są one bardzo proste do napisania, a ich wartość dokumentalna jest wyższa niż koszt produkcji.

T4. Ignorowany test jest wskazaniem niejednoznaczności

Czasami jesteśmy niepewni szczegółów działania, ponieważ wymagania są niejasne. Możemy wyrazić nasze pytanie na temat wymagania w postaci zakomentowanego testu lub testu oznaczonego za pomocą `@Ignore`. Metoda, jaką wybierzemy, zależy od tego, czy niejednoznaczność dotyczy czegoś, co się kompiluje, czy nie.

T5. Warunki graniczne

Szczególną uwagę należy poświęcić warunkom granicznym. Często prawidłowo realizujemy środkową część algorytmu, ale mamy błędy w warunkach granicznych.

T6. Dokładne testowanie pobliskich błędów

Błędy mają tendencję do łączenia się. Gdy znajdziemy błąd w funkcji, mądrze jest wykonać dokładne testy funkcji. Prawdopodobnie okaże się, że ten błąd nie był jedynym.

T7. Wzorce błędów wiele ujawniają

Czasami możemy zdiagnozować problem przez znalezienie wzorca błędów przypadków testowych. Jest to kolejny argument przemawiający za tworzeniem możliwie kompletnych przypadków testowych. Kompletne przypadki testowe, które są uporządkowane w sensowny sposób, pozwalają ujawniać takie wzorce.

Może być to przypadek, gdy nie wszystkie testy są wykonywane z danymi wejściowymi dłuższymi niż pięć znaków. Innym przykładem może być złe wykonywanie wszystkich testów, w których drugi argument jest liczbą ujemną. Czasami tylko zobaczenie wzorów kolorów czerwonego i zielonego na raporcie testów wystarczy nam do znalezienia rozwiązania. Interesujące przykłady można znaleźć w rozdziale 16. przy okazji przedstawiania przykładu klasy `SerialDate`.

T8. Wzorce pokrycia testami wiele ujawniają

Spojrzenie na kod, który jest lub nie jest wykonywany przez testy, daje wskazówki odnośnie do przyczyny błędu testowania.

T9. Testy powinny być szybkie

Powolne testy nie będą wykonywane. Gdy zacznie nam brakować czasu, powolne testy będą usuwane z zestawu. Dlatego *należy robić wszystko*, aby testy były szybkie.

Zakończenie

Trudno stwierdzić, czy ta lista zapachów i heurystyk jest kompletna. W rzeczywistości nie jestem pewien, czy taka lista *kiedykolwiek* będzie kompletna. Jednak kompletność nie powinna być tu celem, ponieważ lista ta *pozwala* na ocenę systemu.

W rzeczywistości ocena systemu powinna być celem i tematem tej książki. Czysty kod nie jest tworzony przez zamieszczoną tu listę zasad. Nikt nie stanie się świetnym programistą przez nauczenie się listy heurystyk. Profesjonalizm i umiejętności biorą się z dyscypliny ich stosowania.

Bibliografia

[Refactoring]: Martin Fowler i inni, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley 1999.

[PRAG]: Andrew Hunt, Dave Thomas, *The Pragmatic Programmer*, Addison-Wesley 2000.

[GOF]: Gamma i inni, *Elements of Reusable Object Oriented Software*, Addison-Wesley 1996.

[Beck97]: Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall 1997.

[Beck07]: Kent Beck, *Implementation Patterns*, Addison-Wesley 2008.

[PPP]: Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall 2002.

[DDD]: Eric Evans, *Domain Driven Design*, Addison-Wesley 2003.

DODATEK A

Współbieżność II

Brett L. Schuchert

DODATEK TEN UZUPEŁNIA rozdział 13., „Współbieżność”. Został on napisany w postaci serii niezależnych tematów, które mogą być czytane w dowolnej kolejności. Aby umożliwić taki sposób czytania, występuje tu kilka powtórzeń między punktami.

Przykład klient-serwer

Wyobraźmy sobie prostą aplikację klient-serwer. Serwer nasłuchuje na gnieździe, oczekując na podłączenie się klienta. Klient podłącza się i wysyła żądanie.

Serwer

Poniżej przedstawiona jest uproszczona wersja aplikacji serwera. Pełny kod źródłowy zaczyna się w dalszej części dodatku, w punkcie „Klient-serwer bez wątków”.

```
ServerSocket serverSocket = new ServerSocket(8009);
while (keepProcessing) {
  try {
    Socket socket = serverSocket.accept();
    process(socket);
  } catch (Exception e) {
    handle(e);
  }
}
```

Ta prosta aplikacja oczekuje na połączenia, przetwarza przychodzące komunikaty i znów oczekuje na przyjście następnego żądania klienta. Poniżej mamy kod klienta podłączającego się do tego serwera:

```
private void connectSendReceive(int i) {
  try {
    Socket socket = new Socket("localhost", PORT);
    MessageUtils.sendMessage(socket, Integer.toString(i));
    MessageUtils.getMessage(socket);
    socket.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Jak szybko działa para klient-serwer? W jaki sposób możemy formalnie opisać jej wydajność? Mamy tu test, który zakłada, że wydajność jest „akceptowalna":

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
  Thread[] threads = createThreads();
  startAllThreadsw(threads);
  waitForAllThreadsToFinish(threads);
}
```

Konfiguracja nie jest tu zamieszczona, aby uprościć nasz test (patrz listing A4. ClientTest.java). Test ten zakłada, że powinien zakończyć się w czasie 10 000 milisekund.

Jest to klasyczny przykład kontroli przepustowości systemu. System ten powinien zakończyć serię żądań klientów w czasie dziesięciu sekund. Jeżeli serwer będzie w stanie przetworzyć poszczególne żądania klienta na czas, test się powiedzie.

Co się stanie, jeżeli test się nie uda? Poza utworzeniem pętli odpytywania zdarzeń nie da się zrobić zbyt wiele z jednym wątkiem w celu przyspieszenia działania tego kodu. Czy zastosowanie wielu wątków rozwiąże problem? Być może, ale musimy wcześniej wiedzieć, gdzie jest marnowany czas. Istnieją dwie możliwości:

- Wejście-wyjście — korzystanie z gniazd, podłączanie się do bazy danych, oczekiwanie na przełączenie pamięci wirtualnej i tak dalej.
- Procesor — obliczenia numeryczne, przetwarzanie wyrażeń regularnych, zbieranie nieużytków i tak dalej.

Zwykle systemy mają kilka takich wąskich gardeł, ale w przypadku danej operacji najczęściej dominuje jedno. Jeżeli kod jest związany z procesorem, to lepszy sprzęt przetwarzający może poprawić wydajność i testy się powiodą. Jednak mamy dostępną tylko określoną liczbę cykli CPU, więc dodanie wątków do problemu związanego z procesorem nie przyspieszy działania programu.

Jeżeli jednak proces jest ograniczany przez wejście-wyjście, to współbieżność może poprawić efektywność. Gdy jedna część systemu oczekuje na operację wejścia-wyjścia, inna część może wykorzystać ten czas oczekiwania do przetworzenia czegoś innego, bardziej efektywnie korzystając z dostępnych zasobów procesora.

Dodajemy wątki

Załóżmy teraz, że test wydajnościowy się nie powiódł. Jak możemy poprawić przepustowość, aby zaliczyć ten test? Jeżeli metoda serwera `process` jest ograniczana przez wejście-wyjście, to możemy w serwerze zastosować wątki (wystarczy zmienić `processMessage`):

```
void process(final Socket socket) {
  if (socket == null)
    return;
  Runnable clientHandler = new Runnable() {
    public void run() {
      try {
        String message = MessageUtils.getMessage(socket);
        MessageUtils.sendMessage(socket, "Processed: " + message);
        closeIgnoringException(socket);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  };
  Thread clientConnection = new Thread(clientHandler);
  clientConnection.start();
}
```

Załóżmy, że zmiana ta spowodowała, iż test zostanie wykonany[1]; kod jest kompletny, zgadza się?

Uwagi na temat serwera

Zaktualizowany serwer wykonuje test w nieco ponad sekundę. Niestety, rozwiązanie to jest trochę naiwne i wprowadza kilka nowych problemów.

Ile wątków może utworzyć nasz serwer? W kodzie nie ma ograniczeń, więc możemy bez trudu dojść do ograniczenia nakładanego przez maszynę wirtualną środowiska Java (JVM). Dla wielu prostych systemów może to wystarczyć. Co jednak, jeżeli system jest przeznaczony do obsługi wielu użytkowników w sieci publicznej? Jeżeli zbyt wielu użytkowników podłączy się w tym samym czasie, system może się zatrzymać.

Jednak pozostawmy na moment problem działania. Przedstawione rozwiązanie ma problemy z czystością i strukturą. Ile odpowiedzialności ma kod serwera? Oto one:

- zarządzanie podłączaniem do gniazd,
- obsługa klientów,
- zasady wątków,
- zasady wyłączenia serwera.

[1] Można to sprawdzić samemu, testując obie wersje kodu. Warto zapoznać się z kodem niekorzystającym z wątków, z punktu „Klient-serwer bez wątków", a także z kodem korzystającym z wątków, z punktu „Klient-serwer z wątkami".

Niestety, wszystkie te odpowiedzialności są umieszczone w funkcji `process`. Dodatkowo kod przekracza wiele różnych poziomów abstrakcji. Dlatego funkcja przetwarzania wymaga podzielenia, pomimo że jest tak mała.

Istnieje kilka potencjalnych powodów zmiany serwera, więc łamie to zasadę pojedynczej odpowiedzialności. Aby zachować czystość systemów współbieżnych, zarządzanie wątkami powinno być umieszczone w niewielu odpowiednio kontrolowanych miejscach. Co więcej, każdy kod zarządzający wątkami powinien nie robić nic poza zarządzaniem wątkami. Dlaczego? Wystarczy powiedzieć, że śledzenie problemów ze współbieżnością jest wystarczająco trudne bez konieczności jednoczesnego rozwiązywania problemów niezwiązanych z nią.

Jeżeli utworzymy osobną klasę dla każdej wymienionej wcześniej odpowiedzialności, w tym dla odpowiedzialności zarządzania wątkami, to gdy zmienimy strategię zarządzania wątkami, będzie ona miała wpływ na mniejszą część kodu i nie będzie przeszkadzać w innych odpowiedzialnościach. Dodatkowo pozwala to łatwiej testować wszystkie inne odpowiedzialności, bez obawy o wielowątkowość. Poniżej przedstawiona jest zmieniona wersja kodu, która działa właśnie w taki sposób:

```
public void run() {
  while (keepProcessing) {
    try {
      ClientConnection clientConnection = connectionManager.awaitClient();
      ClientRequestProcessor requestProcessor
        = new ClientRequestProcessor(clientConnection);
      clientScheduler.schedule(requestProcessor);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  connectionManager.shutdown();
}
```

Teraz wszystkie elementy związane z wątkami znajdują się w jednym miejscu, `clientScheduler`. Jeżeli wystąpią problemy ze współbieżnością, mamy jedno miejsce do sprawdzenia:

```
public interface ClientScheduler {
  void schedule(ClientRequestProcessor requestProcessor);
}
```

Bieżące zasady są łatwe do zaimplementowania:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
  public void schedule(final ClientRequestProcessor requestProcessor) {
    Runnable runnable = new Runnable() {
      public void run() {
        requestProcessor.process();
      }
    };
    Thread thread = new Thread(runnable);
    thread.start();
  }
}
```

Wyizolowanie wszystkich elementów zarządzania wątkami do jednego miejsca pozwala na znacznie łatwiejsze zmienianie sposobu sterowania wątkami. Na przykład przejście na bibliotekę Java 5 Executor wymaga napisania nowej klasy i jej podłączenia (listing A1).

LISTING A1. ExecutorClientScheduler.java

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
  Executor executor;

  public ExecutorClientScheduler(int availableThreads) {
    executor = Executors.newFixedThreadPool(availableThreads);
  }

  public void schedule(final ClientRequestProcessor requestProcessor) {
    Runnable runnable = new Runnable() {
      public void run() {
        requestProcessor.process();
        }
    };
    executor.execute(runnable);
  }
}
```

Zakończenie

Wprowadzanie współbieżności w tym określonym przykładzie demonstruje sposób poprawiania przepustowości systemu oraz jeden sposób kontrolowania przepustowości za pomocą testów. Umieszczenie całego kodu współbieżności w małej liczbie klas jest przykładem zastosowania zasady pojedynczej odpowiedzialności. W przypadku programowania współbieżnego staje się to szczególnie istotne z powodu jego złożoności.

Możliwe ścieżki wykonania

Przyjrzyjmy się jednowierszowej metodzie `incrementValue` napisanej w języku Java, niezawierającej pętli ani instrukcji warunkowych.

```
public class IdGenerator {
  int lastIdUsed;

  public int incrementValue() {
    return ++lastIdUsed;
  }
}
```

Zignorujmy przepełnienie liczby całkowitej i załóżmy, że tylko jeden wątek ma dostęp do jednego obiektu `IdGenerator`. W takim przypadku mamy jedną ścieżkę wykonania i jeden gwarantowany wynik:

- Zwracana wartość jest równa wartości `lastIdUsed`, która jest o jeden większa niż przed wywołaniem tej metody.

Co się stanie, jeżeli użyjemy dwóch wątków i pozostawimy metodę niezmienioną? Jakie są możliwe wyniki, jeżeli każdy wątek jeden raz wywoła `incrementValue`? Ile mamy możliwych ścieżek wykonania? Na początek wyniki (zakładamy, że `lastIdUsed` zaczyna od wartości `93`):

- Wątek 1. otrzymuje wartość `94`, wątek 2. otrzymuje wartość `95`, `lastUsedId` jest równe `95`.
- Wątek 1. otrzymuje wartość `95`, wątek 2. otrzymuje wartość `94`, `lastUsedId` jest równe `95`.
- Wątek 1. otrzymuje wartość `94`, wątek 2. otrzymuje wartość `94`, `lastUsedId` jest równe `94`.

Ostatni wynik, choć zaskakujący, jest możliwy. Aby sprawdzić, w jaki sposób są możliwe te różne wyniki, musimy poznać liczbę możliwych ścieżek wykonania i sposób ich realizacji przez maszynę wirtualną Java.

Liczba ścieżek

Aby obliczyć liczbę możliwych ścieżek wykonania, zaczniemy od wygenerowanego kodu bajtowego. Jeden wiersz kodu Java (`return ++lastIdUsed;`) staje się ośmioma instrukcjami kodu bajtowego. Możliwe jest, że dwa wątki przeplatają wykonanie tych ośmiu instrukcji, w sposób podobny do tasowania kart przez krupiera[2]. Nawet jeżeli w każdej ręce będzie miał osiem kart, to wynikowa liczba przetasowań jest znaczna.

Dla tego prostego przypadku sekwencji N instrukcji, bez pętli i warunków oraz T wątków, całkowita liczba możliwych ścieżek wykonania jest równa:

$$\frac{(NT)!}{N!^T}$$

Wyliczanie możliwych uporządkowań

Metoda pochodzi z e-maila wujka Boba do Bretta:

Przy N krokach i T wątkach mamy razem $T \times N$ kroków. Przed każdym krokiem wykonywane jest przełączenie kontekstu, które pozwala na wybranie spośród T wątków. Każda ścieżka może być reprezentowana jako ciąg cyfr określających przełączenia kontekstu. Jeżeli mamy kroki A i B oraz wątki 1. i 2., sześcioma możliwymi ścieżkami będą 1122, 1212, 1221, 2112, 2121 oraz 2211. Zapisując to w kontekście kroków, uzyskujemy: A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 oraz A2B2A1B1. Dla trzech wątków mamy następujące sekwencje: 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, … .

Ciągi te mają jedną ważną cechę — zawsze musi znajdować się N wystąpień każdego T. Dlatego ciąg 111111 jest nieprawidłowy, ponieważ ma sześć wystąpień 1 i zero wystąpień 2 i 3.

Oczekujemy permutacji N jedynek, N dwójek, … oraz N wystąpień T. Jest to naprawdę permutacja $N \times T$ elementów po $N \times T$ jednocześnie, czyli $(N \times T)!$, ale z usuniętymi powtórzeniami. Musimy tylko policzyć powtórzenia i odjąć je od $(N \times T)!$

[2] Jest to niewielkie uproszczenie. Jednak na potrzeby tej dyskusji możemy użyć takiego uproszczonego modelu.

Mając dwa kroki i dwa wątki, ile znajdziemy powtórzeń? Każdy czterocyfrowy ciąg ma dwie 1 i dwie 2. Każda z tych par może być zamieniona bez zmiany sensu ciągu. Można zamienić jedynki lub dwójki. Mamy więc cztery równorzędne postacie dla każdego ciągu, co oznacza, że mamy tu trzy powtórzenia. Trzy z czterech opcji to powtórzenia albo, inaczej mówiąc, jedna z czterech permutacji nie jest powtórzeniem. 4!×0,25 = 6. Wnioskowanie to wydaje się prawidłowe.

Ile mamy powtórzeń? W przypadku, gdy $N = 2$ i $T = 2$, mogę zamienić jedynki, dwójki lub obie. W przypadku, gdy $N = 2$, a $T = 3$, możemy zamienić jedynki, dwójki, trójki, jedynki i dwójki, jedynki i trójki lub dwójki i trójki. Zamiana jest po prostu permutacją *N*. Załóżmy, że mamy *P* permutacji *N*. Liczba różnych sposobów uporządkowania tych permutacji wynosi $P\times\times T$.

Dlatego liczba możliwych postaci równorzędnych wynosi $N!\times\times T$. Więc liczba ścieżek wynosi $(T\times N)!/(N!\times\times T)$. W naszym przypadku, $T = 2$ i $N = 2$, otrzymamy 6 (24/4).

Dla $N = 2$ oraz $T = 3$ otrzymamy 720/8 = 90.

Dla $N = 3$ oraz $T = 3$ otrzymamy 9!/3^3 = 1680.

Dla naszego prostego kodu Java zawierającego jeden wiersz, który daje osiem wierszy kodu bajtowego, oraz dwóch wątków całkowita liczba możliwych ścieżek wykonania wynosi 12 870. Jeżeli zmienna `lastIdUsed` byłaby typu `long`, to każdy odczyt i zapis obejmowałby dwie operacje zamiast jednej, więc liczba możliwych uporządkowań wyniosłaby 2 704 156.

Co się stanie, jeżeli do tej metody wprowadzimy jedną zmianę?

```
public synchronized void incrementValue() {
  ++lastIdUsed;
}
```

Liczba możliwych ścieżek wykonania byłaby równa 2 dla dwóch wątków i *N!* w ogólnym przypadku.

Kopiemy głębiej

Co to za zaskakujący wynik, jeżeli dwa wątki mogą wywołać jednocześnie tę samą metodę (przed dodaniem `synchronized`) i otrzymamy ten sam wynik numeryczny? Jak to możliwe? Na początek podstawy.

Co jest operacją atomową? Możemy ją zdefiniować jako operację, której nie można przerwać. Na przykład w poniższym kodzie wiersz 5., gdzie jest przypisywana wartość 0 do `lastId`, jest atomowy, ponieważ zgodnie z modelem pamięci Java przypisanie wartości 32-bitowej jest nieprzerywalne.

```
O1: public class Example {
O2:   int lastId;
O3:
```

```
04:   public void resetId() {
05:     value = 0;
06:   }
07:
08:   public int getNextId() {
09:     ++value;
10:   }
11:}
```

Co się stanie, jeżeli zmienimy typ `lastId` na `long`? Czy wiersz 5. jest nadal atomowy? Nie, zgodnie ze specyfikacją JVM. Może być on atomowy na określonym procesorze, ale zgodnie ze specyfikacją JVM przypisanie dowolnej wartości 64-bitowej wymaga dwóch przypisań 32-bitowych. Oznacza to, że pomiędzy pierwszym przypisaniem 32-bitowym a drugim przypisaniem 32-bitowym inny wątek może się wślizgnąć i zmienić jedną z wartości.

Co można powiedzieć o operatorze inkrementacji z wiersza 9.? Operator ten może być przerwany, więc nie jest atomowy. Aby to zrozumieć, przejrzyjmy kod bajtowy obu tych metod.

Zanim będziemy mogli kontynuować, przytoczymy ważne definicje:

- Ramka — każde wywołanie metody wymaga ramki. Ramka zawiera adres powrotu, wszystkie parametry przekazane do metody oraz zmienne lokalne zdefiniowane w metodzie. Jest to standardowa technika używana do definiowania stosu wywołań, który jest wykorzystywany przez nowoczesne języki do podstawowego wywołania metod i funkcji oraz umożliwienia wywoływania rekurencyjnego.
- Zmienna lokalna — dowolna zmienna zdefiniowana w zakresie metody. Wszystkie niestatyczne metody mają co najmniej jedną zmienną, **`this`**, która reprezentuje bieżący obiekt — obiekt, który otrzymał ostatni komunikat (w bieżącym wątku), jaki spowodował wywołanie metody.
- Stos operandów — wiele z instrukcji w maszynie wirtualnej Java posiada parametry. Parametry te są odkładane na stos operandów. Stos jest strukturą danych typu „ostatni wszedł, pierwszy wyszedł" (LIFO).

Poniżej zamieszczony jest kod bajtowy wygenerowany dla `resetId()`.

Mnemonika	Opis	Końcowy stos operandów
`ALOAD 0`	Załadowanie zerowej zmiennej na stos operandów. Co jest zerową zmienną? Jest to `this`, bieżący obiekt. Gdy zostaje wywołana metoda, odbiorca komunikatu, obiekt `Example`, jest umieszczany w tablicy zmiennych lokalnych ramki utworzonej dla wywołania metody. Jest to zawsze pierwsza zmienna umieszczana dla każdej metody instancyjnej.	`this`
`ICONST_0`	Umieszczenie wartości stałej 0 na stosie operandów.	`this, 0`
`PUTFIELD lastId`	Zapamiętanie wartości ze szczytu stosu (czyli 0) w polu obiektu, do którego odwołuje się referencja obiektu ze szczytu stosu, czyli **`this`**.	<pusty>

Gwarantowane jest, że te trzy instrukcje są atomowe, ponieważ pomimo możliwości przerwania wątku po każdej z nich, informacja dla instrukcji `PUTFIELD` (stała 0 ze szczytu stosu oraz odwołanie do `this` poniżej szczytu, jak również wartość pola) nie może być zmieniona przez żaden wątek. Dlatego, gdy nastąpi przypisanie, gwarantowane jest, że w polu zostanie zapisana wartość 0. Operacja jest atomowa. Wszystkie operandy działają na informacjach lokalnych dla metody, więc nie ma interferencji między wieloma wątkami.

Jeżeli te trzy instrukcje są wykonywane przez dziesięć wątków, może wystąpić 4,38679733629e+24 uporządkowań. Jednak jest tylko jeden możliwy wynik, więc różne uporządkowania są bez znaczenia. W tym przypadku ten sam wynik jest gwarantowany również dla wartości `long`. Dlaczego? Wszystkie dziesięć wątków przypisuje stałą wartość. Nawet jeżeli przeplatają się, ostateczny wynik będzie taki sam.

W przypadku operacji ++ w metodzie `getNextId` zaczynają się problemy. Załóżmy, że na początku działania tej metody `lastId` zawiera wartość 42. Poniżej pokazany jest kod bajtowy dla tej nowej metody.

Mnemonika	Opis	Końcowy stos operandów
`ALOAD 0`	Ładuje `this` na stos operandów.	`this`
`DUP`	Kopiuje szczyt stosu. Na stosie operandów mamy teraz dwie kopie `this`.	`this`, `this`
`GETFIELD lastId`	Pobiera wartość pola `lastId` z obiektu wskazywanego przez referencję na szczycie stosu (`this`) i zapisuje wartość z powrotem na stosie.	`this`, 42
`ICONST_1`	Umieszcza stałą całkowitą 1 na stosie.	`this`, 42, 1
`IADD`	Dodaje dwie wartości całkowite na stosie operandów i zapisuje wynik z powrotem na stosie operandów.	`this`, 43
`DUP_X1`	Powiela wartość 43 i umieszcza ją przed `this`.	43, `this`, 43
`PUTFIELD value`	Zapisuje wartość ze szczytu stosu operandów, 43, do pola bieżącego obiektu, reprezentowanego przez drugą od szczytu wartość stosu operandów, czyli `this`.	43
`IRETURN`	Zwraca wartość ze szczytu stosu (tylko tę).	<pusty>

Wyobraźmy sobie przypadek, gdy pierwszy wątek kończy pierwsze trzy instrukcje, aż do `GETFIELD` włącznie, i jest przerywany. Drugi wątek przejmuje sterowanie i wykonuje całą metodę, zwiększając `lastId` o jeden; otrzymamy wartość 43. Następnie pierwszy wątek wznawia pracę tam, gdzie został przerwany, a na jego stosie operandów nadal znajduje się 42, ponieważ w czasie wykonywania `GETFIELD` taka była właśnie wartość `lastId`. Znów dodawana jest wartość jeden i ponownie zapisywany jest wynik 43. Wartość 43 jest zwracana również z pierwszego wątku. W wyniku tego jedna inkrementacja jest utracona, ponieważ pierwszy wątek został przerwany przez drugi.

Oznaczenie metody `getNextId()` jako `synchronized` rozwiązuje problem.

Zakończenie

Szczegółowa znajomość kodu bajtowego nie jest niezbędna do zrozumienia tego, jak wątki mogą się na siebie nakładać. Jeżeli zrozumiemy przedstawiony tu przykład, powinniśmy wiedzieć o możliwości nakładania się dwóch wątków, co jest wystarczającą wiedzą.

Ten prosty przykład miał za zadanie pokazać potrzebę zrozumienia modelu pamięci na tyle, aby wiedzieć, co jest bezpieczne, a co nie. Często spotykaną pomyłką jest założenie, że operator ++ (zarówno pre-, jak i postinkrementujący) jest atomowy. To nieprawda. Oznacza to, że powinniśmy wiedzieć:

- gdzie znajdują się współużytkowane obiekty i wartości,
- który kod może spowodować problemy z równoległym odczytem i zapisem,
- jak chronić się przed wystąpieniem problemów ze współbieżnością.

Poznaj używaną bibliotekę

Biblioteka Executor

Jak jest pokazane na listingu A1, ExecutorClientScheduler.java, biblioteka `Executor`, wprowadzona w Java 5, pozwala na złożone operacje wykorzystujące pule wątków. Klasa ta znajduje się w pakiecie `java.util.concurrent`.

Jeżeli tworzymy wątki i nie korzystamy z pul wątków lub *korzystamy* z własnych, powinniśmy rozważyć wykorzystanie klasy `Executor`. Pozwala to na tworzenie czystszego, łatwiejszego do analizy i mniejszego kodu.

Biblioteka `Executor` zarządza pulą wątków, zmienia ją automatycznie i w razie potrzeby odtwarza wątki. Obsługuje ona obiekty *future*, często stosowaną konstrukcję programowania współbieżnego. Biblioteka `Executor` operuje na klasach implementujących `Runnable` i dodatkowo na klasach implementujących interfejs `Callable`. Interfejs `Callable` jest podobny do `Runnable`, ale pozwala na zwrócenie wyniku, co jest często potrzebne w rozwiązaniach wielowątkowych.

Obiekt *future* jest wygodny, jeżeli w naszym kodzie istnieje potrzeba wykonania wielu niezależnych operacji i oczekiwania na ich zakończenie.

```
public String processRequest(String message) throws Exception {
  Callable<String> makeExternalCall = new Callable<String>() {
    public String call() throws Exception {
      String result = "";
      // Wykonanie zewnętrznego żądania.
      return result;
    }
  };
  Future<String> result = executorService.submit(makeExternalCall);
  String partialResult = doSomeLocalProcessing();
  return result.get() + partialResult;
}
```

W tym przykładzie metoda uruchamia wykonywanie obiektu `makeExternalCall`. Następnie metoda kontynuuje przetwarzanie. W ostatnim wierszu wywołanie `result.get()` powoduje zablokowanie działania do zakończenia *future*.

Rozwiązania nieblokujące

Maszyna wirtualna Java 5 wykorzystuje projekty nowoczesnych procesorów, które obsługują niezawodne aktualizacje nieblokujące. Jako przykład weźmy klasę korzystającą z synchronizacji (a więc i blokowania) do zapewnienia bezpiecznej dla wątków aktualizacji wartości:

```
public class ObjectWithValue {
  private int value;

  public void synchronized incrementValue() { ++value; }
  public int getValue() { return value; }
}
```

Java 5 posiada zestaw nowych klas pozwalających na obsłużenie takich sytuacji — przykładami mogą być `AtomicBoolean`, `AtomicInteger` czy `AtomicReference`, ale dostępnych jest jeszcze kilka innych. Możemy zmienić powyższy kod tak, aby korzystał z rozwiązania nieblokującego:

```
public class ObjectWithValue {
  private AtomicInteger value = new AtomicInteger(0);

  public void incrementValue() {
    value.incrementAndGet();
  }

  public int getValue() {
    return value.get();
  }
}
```

Mimo że wykorzystany jest obiekt zamiast wartości prostej i wysyłany jest komunikat `increment ↪AndGet()` zamiast ++, wydajność działania tej klasy niemal zawsze przewyższa wcześniejszą wersję. W niektórych przypadkach kod ten jest tylko nieznacznie szybszy, ale przypadki, gdy rozwiązanie to jest wolniejsze, praktycznie nie istnieją.

Jak to możliwe? Nowoczesne procesory udostępniają operację zwykle nazywaną *Compare and Swap (CAS)*. Operacja ta jest analogiczna do blokowania optymistycznego w bazie danych, natomiast wersja synchronizowana jest analogiczna do blokowania pesymistycznego.

Słowo kluczowe `synchronized` zawsze nakłada blokadę, nawet jeżeli drugi wątek nie próbuje zaktualizować tej samej blokady. Mimo że wydajność koniecznych blokad z każdą wersją się poprawia, nadal są one kosztowne.

Wersja nieblokująca wychodzi z założenia, że wiele wątków zwykle nie modyfikuje tej samej wartości na tyle często, aby powstały problemy. Zamiast tego efektywnie wykrywa moment wystąpienia takiej sytuacji i powtarza operację do prawidłowego zakończenia aktualizacji. Takie wykrywanie jest niemal zawsze mniej kosztowne niż nakładanie blokady, nawet w przypadku średniego i wysokiego obciążenia.

W jaki sposób realizuje to maszyna wirtualna? Operacja CAS jest atomowa. Logicznie rzecz biorąc, operacja CAS wygląda mniej więcej tak:

```
int variableBeingSet;

void simulateNonBlockingSet(int newValue) {
  int currentValue;
  do {
    currentValue = variableBeingSet
  } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
  if(variableBeingSet == currentValue) {
    variableBeingSet = newValue;
    return currentValue;
  }
  return variableBeingSet;
}
```

Gdy metoda próbuje zaktualizować współużytkowaną zmienną, operacja CAS sprawdza, czy zmienna ta nadal posiada znaną wartość. Jeżeli tak, zmienna jest modyfikowana. Jeżeli nie, zmienna nie jest ustawiana, ponieważ inny wątek wszedł nam w drogę. Metoda wykonująca operację (przy użyciu CAS) zauważa, że zmiana nie została wykonana, i ponawia próbę.

Bezpieczne klasy nieobsługujące wątków

Niektóre klasy z natury nie są bezpieczne dla wątków. Kilka przykładów takich klas przedstawiono poniżej:

- `SimpleDateFormat`,
- połączenia z bazami danych,
- kontenery w `java.util`,
- serwlety.

Niektóre klasy kolekcji posiadają pojedyncze metody bezpieczne dla wątków. Jednak każda operacja wymagająca wywołania więcej niż jednej metody nie jest bezpieczna. Na przykład, gdy nie chcemy zamieniać elementu z `HashTable`, jeżeli już się tam znajduje, możemy napisać następujący kod:

```
if(!hashTable.containsKey(someKey)) {
  hashTable.put(someKey, new SomeValue());
}
```

Każda z tych metod jest bezpieczna dla wątków. Jednak inny wątek może dodać wartość pomiędzy wywołaniami `containsKey` oraz `put`. Istnieje kilka możliwości rozwiązania tego problemu:

- Wcześniejsze zablokowanie `HashTable` i upewnienie się, że wszyscy użytkownicy zrobili to samo — blokowanie z poziomu klienta:

```
synchronized(map) {
  if(!map.conainsKey(key))
    map.put(key,value);
}
```

- Opakowanie `HashTable` we własny obiekt i użycie innego API — blokowanie z poziomu serwera wykorzystujące wzorzec adapter:

```
public class WrappedHashtable<K, V> {
  private Map<K, V> map = new Hashtable<K, V>();

  public synchronized void putIfAbsent(K key, V value) {
    if (map.containsKey(key))
      map.put(key, value);
  }
}
```

- Zastosowanie kolekcji bezpiecznych dla wątków:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer,String>();
map.putIfAbsent(key, value);
```

Kolekcje z pakietu `java.util.concurrent` posiadają metody, takie jak `putIfAbsent()`, pozwalające obsłużyć takie operacje.

Zależności między metodami mogą uszkodzić kod współbieżny

Poniżej przedstawiony jest prosty przykład wprowadzenia zależności między modułami:

```
public class IntegerIterator implements Iterator<Integer>
  private Integer nextValue = 0;

  public synchronized boolean hasNext() {
    return nextValue < 100000;
  }

  public synchronized Integer next() {
    if (nextValue == 100000)
      throw new IteratorPastEndException();
    return nextValue++;
  }

  public synchronized Integer getNextValue() {
    return nextValue;
  }
}
```

Poniżej mamy kod wykorzystujący tę klasę `IntegerIterator`:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
  int nextValue = iterator.next();
  // Wykonanie operacji na nextValue.
}
```

Jeżeli kod ten jest wykonywany przez jeden wątek, nie będzie żadnych problemów. Co się stanie, jeżeli dwa wątki będą współużytkowały jeden obiekt `IntegerIterator` przy takim założeniu, że każdy wątek przetwarza pobrane wartości, ale każdy z elementów na liście będzie przetwarzany tylko raz? W większości przypadków nie stanie się nic złego; wątki będą w zgodzie współużytkowały listę, przetwarzały elementy uzyskane za pomocą iteratora i kończyły pracę, gdy iterator nie będzie

miał kolejnych elementów. Jednak istnieje niewielka szansa, że na końcu iteracji dwa wątki będą interferowały ze sobą i mogą spowodować, że jeden wątek wyjdzie poza iterator i zgłosi wyjątek.

Problem ten jest następujący: wątek 1. wywołuje `hasNext()`, która zwraca `true`. Wątek 1. zostaje wywłaszczony, a wątek 2. wykonuje to samo wywołanie, nadal otrzymując `true`. Wątek 2. wywołuje metodę `next()`, która zwraca oczekiwaną wartość, ale efektem ubocznym jej działania jest zmiana wartości zwracanej przez `hasNext()` na `false`. Wątek 1. wznawia pracę i zakłada, że `hasNext()` nadal ma wartość `true`, więc wywołuje `next()`. Mimo że poszczególne metody są synchronizowane, klient używa ***dwóch*** metod.

Jest to rzeczywisty problem i takie przykłady często zdarzają się we współbieżnym kodzie. W tej sytuacji problem jest szczególnie subtelny, ponieważ jedynym miejscem, w którym może zdarzyć się błąd, jest ostatni krok iteratora. Jeżeli wątki tam właśnie się spotkają, to jeden z nich wychodzi poza iterator. Tego rodzaju błędy zdarzają się najczęściej już po produkcyjnym wdrożeniu systemu i są bardzo trudne do wyśledzenia.

Mamy wtedy trzy możliwości:

- tolerować awarię,
- rozwiązać problem przez zmianę klienta — blokowanie na kliencie,
- rozwiązać problem przez zmianę serwera, co dodatkowo wymaga zmian w kliencie — blokowanie na serwerze.

Tolerowanie awarii

Czasami można tak zorganizować pracę, aby takie awarie nie powodowały problemów. Na przykład klient może przechwycić wyjątek i wyczyścić obiekt. Oczywiście, jest to kiepskie rozwiązanie. Jest podobne do wyczyszczenia wycieków pamięci przez przeładowanie systemu o północy.

Blokowanie na kliencie

Aby `IntegerIterator` mógł działać prawidłowo dla wielu wątków, można zmienić tego klienta (i wszystkie inne) w następujący sposób:

```
IntegerIterator iterator = new IntegerIterator();
while (true) {
  int nextValue;
  synchronized (iterator) {
    if (!iterator.hasNext())
      break;
    nextValue = iterator.next();
  }
  doSometingWith(nextValue);
}
```

Każdy klient wprowadza blokadę przez użycie słowa kluczowego `synchronized`. Takie powtórzenie narusza zasadę DRY, ale może być niezbędne, jeżeli kod korzysta z zewnętrznych narzędzi, które nie są bezpieczne dla wątków.

Strategia ta jest ryzykowna, ponieważ wszyscy programiści korzystający z serwera muszą pamiętać o nałożeniu blokady przed jego użyciem i odblokowaniu po zakończeniu. Wiele (naprawdę wiele!) lat temu pracowałem nad systemem, w którym została zastosowana strategia blokowania zasobów na kliencie. Zasób był używany w setkach różnych miejsc w kodzie. Jeden biedny programista zapomniał o zablokowaniu zasobu w jednym z tych miejsc.

System był wieloterminalowym systemem ze współużytkowaniem czasu, obsługującym księgowość dla związku kierowców ciężarówek Local 705. Komputer znajdował się w pokoju z podniesioną podłogą i klimatyzacją, 80 kilometrów na północ od siedziby Local 705. W siedzibie zainstalowanych było kilkanaście stanowisk wprowadzania danych, na których urzędnicy wprowadzali dokumenty do systemu. Terminale były podłączone do komputera z użyciem dedykowanej linii telefonicznej i półdupleksowych modemów o prędkości 600 b/s (to było bardzo, *bardzo* dawno).

Mniej więcej raz dziennie jeden z terminali „zawieszał się". Nie można było określić żadnych prawidłowości — blokada nie wykazywała żadnych preferencji co do terminala ani określonego czasu. Wyglądało to tak, jakby ktoś rzucał kostkami, wybierając czas i terminal do zablokowania. Czasami blokowało się kilka terminali. W niektórych dniach nie było żadnych zawieszeń.

Na początku jedynym rozwiązaniem było przeładowanie. Jednak operacja ta była trudna do skoordynowania. Musieliśmy zadzwonić do centrali i poczekać na zakończenie pracy na wszystkich terminalach. Wtedy mogliśmy wyłączyć system i ponownie go uruchomić. Jeżeli ktoś robił coś ważnego, co trwało godzinę lub dwie, zablokowany terminal musiał pozostać bezużyteczny.

Po kilku tygodniach debugowania odkryliśmy, że przyczyną był licznik bufora cyklicznego, który tracił synchronizację ze wskaźnikiem. Bufor ten kontrolował wyjście na terminal. Wartość wskaźnika wskazywała, że bufor był pusty, ale licznik wskazywał, że był pełny. Ponieważ był pusty, nie było nic do wyświetlania, ale ponieważ był jednocześnie pełny, nic nie można było dodać do bufora wyświetlania na ekranie.

Wiedzieliśmy więc, dlaczego terminal się blokował, ale nie wiedzieliśmy, dlaczego bufor cykliczny rozsynchronizowywał się. Dodaliśmy więc modyfikację pozwalającą obejść problem. Możliwe było odczytywanie stanu przełączników na panelu kontrolnym komputera (to było bardzo, bardzo, *bardzo* dawno). Napisaliśmy małą funkcję przechwytującą, która w momencie przełączenia jednego z przełączników wyszukiwała bufory cykliczne, które były jednocześnie pełne i puste. Jeżeli go znalazła, ustawiała bufor na pusty. *Voila!* Zablokowany terminal ponownie zaczynał wyświetlać dane.

Nie musieliśmy więc restartować systemu po zablokowaniu terminala. Wystarczył telefon od klienta, a ktoś podchodził do komputera i naciskał przycisk.

Czasami jednak Local 705 pracował w weekendy, a my nie. Dodaliśmy więc funkcję do harmonogramu, która sprawdzała wszystkie bufory cykliczne co minutę i jeżeli któryś był jednocześnie pełny i pusty, resetowała go. Dzięki temu terminale odblokowywały się, zanim ktoś z Local podszedł do telefonu.

Po kilku kolejnych tygodniach przeglądania strona po stronie monolitycznego kodu asemblera znaleźliśmy przyczynę. Wykonaliśmy obliczenia i wyliczyliśmy, że częstotliwość blokowania była spójna z jednym niechronionym użyciem bufora cyklicznego. Musieliśmy więc tylko znaleźć jedno

nieprawidłowe użycie. Niestety, było to bardzo dawno temu i nie mieliśmy narzędzi wyszukujących ani wzajemnych odwołań, ani też innych automatycznych podpowiedzi. Musieliśmy po prostu ślęczeć nad listingami.

Tej chłodnej zimy 1971 roku w Chicago nauczyłem się czegoś ważnego. Blokowanie na kliencie jest do niczego.

Blokowanie na serwerze

Powtórzenia mogą być usunięte przez wprowadzenie do klasy `IntegerIterator` następujących zmian:

```
public class IntegerIteratorServerLocked {
  private Integer nextValue = 0;

  public synchronized Integer getNextOrNull() {
    if (nextValue < 100000)
      return nextValue++;
    else
      return null;
  }
}
```

Musi zmienić się również kod klienta:

```
while (true) {
  Integer nextValue = iterator.getNextOrNull();
  if (next == null)
    break;
  // Wykonanie operacji na nextValue.
}
```

W tym przypadku tak zmieniamy API naszej klasy, aby obsługiwało wielowątkowość[3]. Klient musi skontrolować wartość `null`, zamiast korzystać z `hasNext()`.

Powinniśmy korzystać z blokowania na serwerze z następujących powodów:

- Zmniejsza powtarzanie kodu — blokowanie na kliencie wymusza prawidłowe blokowanie na każdym kliencie. Przeniesienie kodu blokującego na serwer pozwala, aby klienty korzystały z obiektów bez konieczności pisania dodatkowego kodu blokującego.
- Pozwala osiągnąć lepszą wydajność — w przypadku instalacji jednowątkowej pozwala na zamianę serwera bezpiecznego dla wątków na serwer bez obsługi wątków, dzięki czemu można pozbyć się wszystkich narzutów.
- Zmniejsza możliwość wystąpienia błędu — niemożliwe jest spowodowanie awarii przez jednego programistę, który zapomniał o zastosowaniu prawidłowej blokady.
- Wymusza jedną zasadę — zasady są w jednym miejscu, na serwerze, a nie w wielu miejscach, na każdym kliencie.

[3] W rzeczywistości interfejs `Iterator` jest z natury niebezpieczny dla wątków. Nigdy nie był projektowany do pracy na wielu wątkach, więc nie powinno to być niespodzianką.

- Zmniejsza zasięg współużytkowanych zmiennych — klient nie jest ich świadom, jak również sposobu, w jaki są blokowane. Wszystko jest ukryte w serwerze. Jeżeli wystąpi błąd, liczba miejsc do sprawdzenia jest mniejsza.

Co możemy zrobić, jeżeli nie mamy wpływu na kod serwera?

- Skorzystać z wzorca adaptera do zmiany API i dodania blokowania.

```
public class ThreadSafeIntegerIterator {
  private IntegerIterator iterator = new IntegerIterator();

  public synchronized Integer getNextOrNull() {
    if(iterator.hasNext())
      return iterator.next();
    return null;
  }
}
```

- Jeszcze lepszym rozwiązaniem jest użycie kolekcji z rozszerzonym interfejsem, bezpiecznym dla wątków.

Zwiększanie przepustowości

Załóżmy, że musimy skorzystać z sieci i odczytać zawartość zbioru stron z listy adresów URL. Gdy strona jest odczytywana, analizujemy ją i zbieramy pewne statystyki. Po przeczytaniu wszystkich stron wyświetlamy raport podsumowujący.

Poniższa klasa zwraca zawartość strony o podanym adresie URL.

```
public class PageReader {
//...
  public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);
    try {
      httpClient.executeMethod(method);
      String response = method.getResponseBodyAsString();
      return response;
    } catch (Exception e) {
      handle(e);
    } finally {
      method.releaseConnection();
    }
  }
}
```

Następna klasa jest iteratorem, który udostępnia zawartość stron na podstawie iteratora adresów URL:

```
public class PageIterator {
  private PageReader reader;
  private URLIterator urls;

  public PageIterator(PageReader reader, URLIterator urls) {
    this.urls = urls;
    this.reader = reader;
  }

  public synchronized String getNextPageOrNull() {
```

```
      if (urls.hasNext())
        getPageFor(urls.next());
      else
        return null;
    }

    public String getPageFor(String url) {
      return reader.getPageFor(url);
    }
  }
```

Obiekt `PageIterator` może być współużytkowany przez rożne wątki, z których każdy korzysta z własnego obiektu `PageReader` do odczytywania i analizowania stron uzyskanych z iteratora.

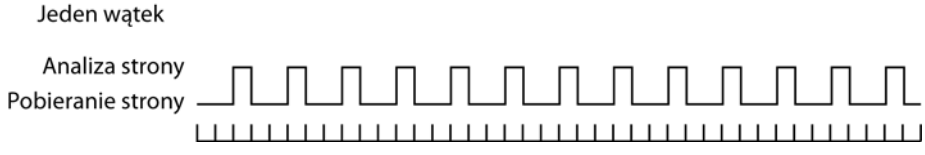
Należy zauważyć, że bloki `synchronized` są bardzo małe. Zawierają wyłącznie sekcje krytyczne głęboko wewnątrz `PageIterator`. Zawsze lepiej synchronizować możliwe małe fragmenty, niż obejmować synchronizacją wszystko, co jest możliwe.

Obliczenie przepustowości jednowątkowej

Wykonajmy teraz proste obliczenia. Na nasze potrzeby załóżmy następujące wielkości:

- Czas wejścia-wyjścia potrzebny do pobrania strony (średnio): 1 sekunda.
- Czas przetwarzania całej strony (średnio): 0,5 sekundy.
- Operacje wejścia-wyjścia wymagają 0 procent czasu procesora, natomiast przetwarzanie 100 procent.

Dla N stron przetwarzanych przez jeden wątek całkowity czas wykonania wynosi 1,5 sekundy × *N*. Na rysunku A1 przedstawione jest przetwarzanie 13 stron, czyli około 19,5 sekundy.

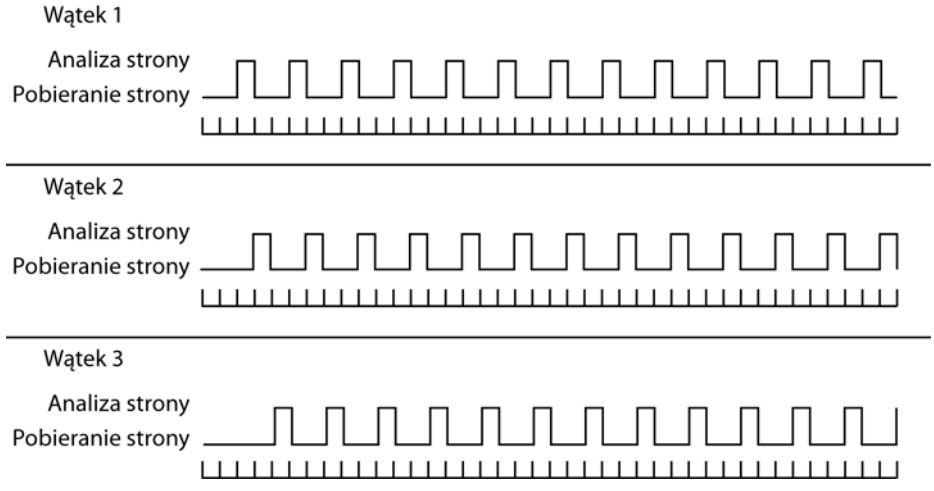


RYSUNEK A1. Jeden wątek

Obliczenie przepustowości wielowątkowej

Jeżeli możliwe jest pobieranie stron w dowolnej kolejności i ich niezależne przetwarzanie, to możliwe jest użycie wielu wątków do zwiększenia przepustowości. Co się stanie, gdy użyjemy trzech wątków? Ile stron przetworzymy w tym samym czasie?

Jak jest pokazane na rysunku A2, rozwiązanie wielowątkowe pozwala na to, aby przetwarzanie stron związane z procesorem przeplatało się z odczytem stron związanym z operacją wejścia-wyjścia. W idealnym przypadku oznacza to, że procesor będzie w pełni wykorzystany. Każde jednosekundowe pobranie będzie przeplatało się z analizą dwóch stron. Dlatego możemy przetwarzać dwie strony na sekundę, czyli trzy razy więcej niż przepustowość rozwiązania jednowątkowego.



RYSUNEK A2. Trzy wątki współbieżne

Zakleszczenie

Wyobraźmy sobie aplikację WWW z dwoma pulami zasobów o skończonym rozmiarze:

- Pula połączeń do bazy danych dla pracy lokalnej przy przetwarzaniu danych.
- Pula połączeń MQ do repozytorium nadrzędnego.

Załóżmy, że w aplikacji mamy dwie operacje — utworzenie i aktualizację:

- Utworzenie — uzyskanie połączenia z głównym repozytorium i bazą danych. Wymiana danych z usługą głównego repozytorium i zapisanie danych w lokalnej bazie danych.
- Aktualizacja — uzyskanie połączenia z bazą danych, a następnie z głównym repozytorium. Odczytanie danych z lokalnej bazy danych i wysłanie ich do głównego repozytorium.

Co się stanie, jeżeli liczba użytkowników będzie przekraczała wielkość pul? Załóżmy, że każda pula ma wielkość równą 10.

- Dziesięciu użytkowników próbuje użyć opcji utworzenia, więc wszystkie (dziesięć) połączenia do bazy danych są zajęte, a każdy wątek jest przerywany po uzyskaniu dostępu do bazy danych, ale przed uzyskaniem połączenia z głównym repozytorium.
- Dziesięciu użytkowników próbuje użyć opcji aktualizacji, więc wszystkie połączenia z głównym repozytorium są zajęte, a każdy wątek jest przerywany po uzyskaniu dostępu do głównego repozytorium, ale przed uzyskaniem połączenia z bazą danych.
- Teraz wszystkie wątki „tworzące” muszą poczekać na uzyskanie połączenia z repozytorium, a jednocześnie dziesięć wątków „aktualizujących” musi poczekać na uzyskanie połączenia z bazą danych.
- Zakleszczenie. System nigdy nie ruszy dalej.

Może się to wydawać mało prawdopodobną sytuacją, ale kto chciałby mieć system, który każdego tygodnia zupełnie przestaje odpowiadać? Kto chce debugować system, mając objawy tak trudne do powtórzenia? Jeżeli tego rodzaju problemy zdarzają się po wdrożeniu, to ich rozwiązanie ciągnie się tygodniami.

Typowym „rozwiązaniem" jest dodanie instrukcji debugujących, które pomagają określić przyczyny problemu. Oczywiście instrukcje debugujące zmieniają kod na tyle, że zakleszczenia pojawiają się w innej sytuacji i znów trzeba czekać miesiące na ich wystąpienie[4].

Aby naprawdę rozwiązać problem zakleszczenia, musimy zrozumieć jego przyczynę. Istnieją cztery warunki wymagane do wystąpienia zakleszczenia:

- wzajemne wykluczanie,
- blokowanie i oczekiwanie,
- brak wywłaszczania,
- cykliczne oczekiwanie.

Wzajemne wykluczanie

Wzajemne wykluczanie występuje, gdy wiele wątków musi użyć tych samych zasobów, a te zasoby:

- nie mogą być używane przez wiele wątków jednocześnie,
- są ograniczone.

Często spotykanym przykładem takich zasobów są połączenia z bazą danych, otwarte do zapisu pliki, blokady rekordów lub semafory.

Blokowanie i oczekiwanie

Gdy wątek uzyska zasób, nie zwalnia go do momentu uzyskania pozostałych zasobów wymaganych do zakończenia zadania.

Brak wywłaszczania

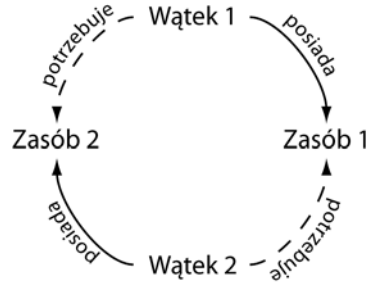
Jeden wątek nie może zabrać zasobów innemu wątkowi. Gdy wątek zablokuje zasób, drugi wątek może skorzystać z niego dopiero po jego zwolnieniu przez wątek blokujący.

Cykliczne oczekiwanie

Jest to nazywane również *śmiertelnym objęciem*. Wyobraźmy sobie dwa wątki, T1 i T2, oraz dwa zasoby, R1 i R2. T1 posiada R1, a T2 posiada R2. T1 wymaga również R2, a T2 wymaga również R1. Daje to w wyniku sytuację, której schemat jest przedstawiony na rysunku A3.

---

[4] Na przykład ktoś dodaje rejestrowanie diagnostyczne i problem „znika". Kod debugujący „naprawia" problem, więc jest pozostawiany w systemie.



RYSUNEK A3.

Wszystkie cztery warunki muszą być spełnione, aby doszło do zakleszczenia.

Zapobieganie wzajemnemu wykluczaniu

Jedną ze strategii unikania zakleszczenia jest zapobieganie warunkom wzajemnego wykluczania. Można to zrobić przez:

- wykorzystywanie zasobów pozwalających na jednoczesne używanie, na przykład `Atomic ↪Integer`;
- zwiększenie liczby zasobów, aby była równa lub przekraczała liczbę rywalizujących wątków;
- sprawdzanie, czy wszystkie zasoby są wolne przed zablokowaniem któregokolwiek.

Niestety, większość zasobów jest ograniczona i nie pozwalają one na jednoczesne wykorzystywanie. Dodatkowo często drugi zasób jest określany na podstawie wyników działania na pierwszym. Jednak nie należy się zrażać, pozostały jeszcze trzy warunki.

Zapobieganie blokowaniu i oczekiwaniu

Można również wyeliminować zakleszczenia, jeżeli nie będziemy oczekiwać na zwolnienie blokady. Należy sprawdzić każdy zasób przed jego zablokowaniem, zwolnić wszystkie zasoby i zacząć od nowa, jeżeli jeden z nich jest zajęty.

Rozwiązanie to wprowadza jednak kilka potencjalnych problemów:

- Zagłodzenie — jeden wątek nie jest w stanie uzyskać potrzebnych zasobów (na przykład jest to unikatowa kombinacja zasobów, która rzadko jest dostępna).
- Uwięzienie — kilka wątków może wpaść w dynamiczne zakleszczenie, gdy wszystkie zajmują jeden zasób, a następnie zwalniają go, i tak w nieskończoność. Jest to szczególnie prawdopodobne dla uproszczonych algorytmów harmonogramowania CPU (w urządzeniach wbudowanych lub prostych własnoręcznie napisanych algorytmach równoważenia wątków).

Obie te sytuacje mogą obniżać przepustowość. Wynikiem pierwszej jest niskie użycie procesora, natomiast drugiej — bezużytecznie wysokie użycie procesora.

Mimo że strategia ta wygląda na mało efektywną, jest lepsza niż nic. Posiada swoje zalety i może być niemal zawsze zastosowana, gdy wszystko inne zawiedzie.

Umożliwienie wywłaszczania

Inną strategią zapobiegania zakleszczeniom jest umożliwienie wątkom zabierania zasobów innym wątkom. Jest to zwykle realizowane przez prosty mechanizm żądań. Gdy wątek wykryje, że zasób jest zajęty, prosi właściciela o jego zwolnienie. Jeżeli właściciel również oczekuje na inny zasób, zwalnia wszystkie i zaczyna od nowa.

Jest to podobne do poprzedniego podejścia, ale pozwala, aby wątek oczekiwał na zasób. Zmniejsza to liczbę restartów. Trzeba jednak pamiętać, że zarządzanie tymi żądaniami może być kłopotliwe.

Zapobieganie oczekiwaniu cyklicznemu

Jest to najczęściej stosowane podejście do zapobiegania zakleszczeniom. W przypadku większości systemów nie wymaga niczego poza uzgodnieniem przez wszystkie strony prostej konwencji.

W wcześniej przedstawionym przykładzie z wątkiem 1., wymagającym zasobów 1. i 2., oraz wątkiem 2., wymagającym zasobów 2. i 1., zwykłe wymuszenie na wątkach 1. i 2. blokowania zasobów w tej samej kolejności spowoduje, że oczekiwanie cykliczne będzie niemożliwe.

Co więcej, jeżeli wszystkie wątki będą stosowały globalne uporządkowanie zasobów i wszystkie będą przydzielać zasoby w tej kolejności, zakleszczenia będą niemożliwe. Jak w przypadku pozostałych strategii, może to powodować problemy:

- Kolejność pobierania może nie być zgodna z kolejnością używania; zasób zajęty na początku może nie być używany aż do końca procesu. Może to powodować dłuższe zajęcie zasobów, niż są potrzebne.
- Czasami nie można wywnioskować kolejności przydzielania zasobów. Jeżeli identyfikator kolejnego zasobu pochodzi z operacji wykonanej na pierwszym, to kolejność może nie być właściwa.

Istnieje wiele sposobów unikania zakleszczeń. Niektóre prowadzą do zagłodzenia, natomiast inne do silnego obciążenia procesora i zwiększenia czasów odpowiedzi. NMDO[5]!

Izolowanie części rozwiązania odpowiedzialnych za wątki pozwalające na dostrajanie i eksperymentowanie jest efektywnym sposobem uzyskania wiedzy pozwalającej na wybranie najlepszej strategii.

[5] Nie ma darmowych obiadów.

Testowanie kodu wielowątkowego

Jak napisać test pokazujący, że poniższy kod zawiera błąd?

```
01: public class ClassWithThreadingProblem {
02:   int nextId;
03:
04:   public int takeNextId() {
05:     return nextId++;
06:   }
07: }
```

Poniżej zamieszczony jest opis testu, który udowadnia, że w kodzie jest błąd:

- Zapamiętaj bieżącą wartość `nextId`.
- Utwórz dwa wątki, które jednocześnie wywołują `takeNextId()`.
- Sprawdź, czy `nextId` ma wartość o dwa większą niż na początku.
- Uruchamiaj test do momentu, gdy `nextId` będzie zwiększone o jeden zamiast dwa.

Test taki jest zamieszczony na listingu A2.

LISTING A2. ClassWithThreadingProblemTest.java

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:      @Test
09:      public void twoThreadsShouldFailEventually() throws Exception {
10:          final ClassWithThreadingProblem classWithThreadingProblem
                 = new ClassWithThreadingProblem();
11:
12:         Runnable runnable = new Runnable() {
13:             public void run() {
14:                 classWithThreadingProblem.takeNextId();
15:             }
16:         };
17:
18:         for (int i = 0; i < 50000; ++i) {
19:             int startingId = classWithThreadingProblem.lastId;
20:             int expectedResult = 2 + startingId;
21:
22:             Thread t1 = new Thread(runnable);
23:             Thread t2 = new Thread(runnable);
24:             t1.start();
25:             t2.start();
26:             t1.join();
27:             t2.join();
28:
29:             int endingId = classWithThreadingProblem.lastId;
30:
31:             if (endingId != expectedResult)
32:                 return;
33:         }
```

```
34:
35:         fail("Powinien pokazać problem z wątkami, ale się to nie udało.");
36:     }
37: }
```

Wiersz	Opis
10	Utworzenie obiektu `ClassWithThreadingProblem`. Musieliśmy tu użyć słowa kluczowego `final`, ponieważ używamy go poniżej wewnętrznej klasy anonimowej.
12 – 16	Utworzenie anonimowej klasy wewnętrznej, która korzysta z jednego obiektu `ClassWithThreadingProblem`.
18	Uruchomienie tego kodu „odpowiednią" liczbę razy w celu pokazania, że kod zawiedzie, ale nie tak dużo, aby test „wykonywał się zbyt długo". Jest to kwestia wyważenia; nie chcemy oczekiwać zbyt długo na pokazanie awarii. Ustalenie tej liczby jest trudne — choć później pokażemy, że można znacznie ją zmniejszyć.
19	Zapamiętanie wartości początkowej. Test ten próbuje udowodnić, że kod w `ClassWith` ↪`ThreadingProblem` jest nieprawidłowy. Jeżeli ten test zostanie wykonany, udowodnimy, że kod zawiera błąd. Jeżeli test zawiedzie, nie będzie w stanie pokazać błędu w kodzie.
20	Oczekujemy, że końcowa wartość będzie o dwa większa niż bieżąca wartość.
22 – 23	Utworzenie dwóch wątków, oba są utworzone w wierszach 12. – 16. Daje to nam potencjalnie dwa wątki próbujące użyć jednego obiektu `ClassWithThreadingProblem`, wpływające wzajemnie na siebie.
24 – 25	Upewnienie się, że dwa wątki są gotowe do uruchomienia.
26 – 27	Oczekiwanie na zakończenie obu wątków przed sprawdzeniem wyników.
29	Zapamiętanie aktualnej wartości końcowej.
31 – 32	Czy wartość `endingId` różni się od oczekiwanej? Jeżeli tak, kończymy test — udowodniliśmy, że kod zawiera błąd. Jeżeli nie, próbujemy jeszcze raz.
35	Jeżeli doszliśmy do tego miejsca, nie jesteśmy w stanie udowodnić w „rozsądnym" czasie, że kod produkcyjny zawiera błąd; nasz test zawiódł. Albo kod nie zawiera błędu, albo nie wykonaliśmy wystarczająco dużo iteracji, aby wystąpiła sytuacja błędna.

Test ten konfiguruje warunki wystąpienia problemu równoległej aktualizacji. Jednak problem występuje tak rzadko, że w większości przypadków test go nie wykryje.

W rzeczywistości, aby wykryć błąd, musimy ustawić liczbę iteracji na ponad milion. Nawet wtedy w dziesięciu uruchomieniach pętli wykonującej 1 000 000 iteracji problem występuje tylko raz. Oznacza to, że aby uzyskać właściwe wyniki, powinniśmy ustawić liczbę iteracji na ponad sto milionów. Na jak długie czekanie jesteśmy przygotowani?

Nawet jeżeli ustawimy test na niezawodne wykrywanie błędu na jednym komputerze, prawdopodobnie będziemy musieli zrobić to ponownie, aby wykryć błąd na innym komputerze, innym systemie operacyjnym lub wersji JVM.

Pamiętajmy, że jest to *prosty* problem. Jeżeli nie możemy łatwo zademonstrować błędu w tym przypadku, to jak wykryć naprawdę skomplikowane problemy?

Jakie podejście możemy przyjąć, aby zademonstrować taki prosty błąd? Co ważniejsze, jak pisać testy, które demonstrują błędy w bardziej złożonym kodzie? W jaki sposób wykryć błędy w kodzie, gdy nie wiemy, gdzie trzeba ich szukać?

Poniżej przedstawiono kilka pomysłów.

- **Testowanie Monte Carlo**. Testy należy pisać elastycznie, aby mogły być dostrajane. Następnie należy je wykonywać w pętli — na przykład na serwerze testowym — losowo zmieniając wartości dostrajające. Jeżeli kiedykolwiek test się nie powiedzie, to w kodzie jest błąd. Należy zapewnić, że testy te zostaną napisane możliwie wcześnie, aby serwer testowy zaczął je wykonywać możliwie szybko. Przy okazji należy pamiętać o precyzyjnym zarejestrowaniu sytuacji, w której test się nie powiódł.
- Testy należy przeprowadzać na każdej platformie docelowej. Wielokrotnie. Ciągle. Im dłużej testy działają bezbłędnie, tym bardziej prawdopodobne jest, że kod produkcyjny jest prawidłowy lub testy nie są w stanie ujawnić problemu.
- Testy powinny być uruchamiane na komputerze o różnym obciążeniu. Jeżeli możemy zasymulować obciążenie zbliżone do występującego w środowisku produkcyjnym, powinniśmy to zrobić.

Jednak nawet jeżeli wykonamy wszystkie te operacje, nadal nie mamy zbyt dużej szansy na znalezienie problemów wielowątkowości w naszym kodzie. W większości przypadków występują one w małych sekcjach i zdarzają się raz na miliard wykonań. Takie problemy są przekleństwem złożonych systemów.

Narzędzia wspierające testowanie kodu korzystającego z wątków

W firmie IBM zostało wyprodukowane narzędzie o nazwie ConTest[6]. Pozwala ono na instrumentację klas w taki sposób, aby kod niebezpieczny dla wątków z większym prawdopodobieństwem wygenerował błąd.

Nie mamy żadnych bezpośrednich związków z firmą IBM ani zespołem produkującym ConTest. Nasi koledzy nam go pokazali. Już po kilku minutach używania tego programu zauważyliśmy ogromną poprawę możliwości znalezienia błędów wielowątkowości.

Sposób wykorzystania programu ConTest jest następujący:

- Napisanie testów i kodu produkcyjnego. Testy powinny być tak zaprojektowane, aby symulować wielu użytkowników przy różnym obciążeniu, tak jak wspomnieliśmy to wcześniej.
- Instrumentacja kodu testów i kodu produkcyjnego za pomocą ConTest.
- Uruchomienie testów.

Gdy przetworzyliśmy nasz kod za pomocą programu ConTest, współczynnik trafień wzrósł z około jednej awarii na dziesięć milionów iteracji do około jednej awarii na *trzydzieści* iteracji. Po instrumentacji wartości licznika pętli dla kilku kolejnych wykonań wynosiły: 13, 23, 0, 54, 16,

[6] *http://www.haifa.ibm.com/projects/verification/contest/index.html*

14, 6, 69, 107, 49, 2. Jasno widać, że klasy po instrumentacji znacznie wcześniej i bardziej niezawodnie pozwalały wykryć błędy.

Zakończenie

Rozdział ten był krótką przejażdżką przez ogromne i zdradzieckie terytorium programowania współbieżnego. Co najwyżej zarysowaliśmy kilka zagadnień. Naszym celem było skupienie się na dyscyplinie zachowania czystości kodu, ale jeżeli zamierzamy pisać systemy współbieżne, powinniśmy nauczyć się znacznie więcej. Na początek polecamy świetną książkę Dougha Lea *Concurrent Programming in Java: Design Principles and Patterns*[7].

W tym rozdziale zajęliśmy się współbieżną aktualizacją oraz dyscypliną czystej synchronizacji i blokowania. Wspominaliśmy o sposobie poprawiania przepustowości systemów związanych z wejściem-wyjściem za pomocą wątków oraz pokazaliśmy techniki osiągania takiej poprawy. Mówiliśmy o zakleszczeniach i sposobach ich unikania. Na koniec przedstawiliśmy strategie ujawniania problemów z wielowątkowością przez instrumentację kodu.

Samouczek. Pełny kod przykładów

Klient-serwer bez wątków

LISTING A3. Server.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
  ServerSocket serverSocket;
  volatile boolean keepProcessing = true;

  public Server(int port, int millisecondsTimeout) throws IOException {
  serverSocket = new ServerSocket(port);
  serverSocket.setSoTimeout(millisecondsTimeout);
  }

  public void run() {
    System.out.printf("Serwer uruchamia się\n");

    while (keepProcessing) {
      try {
        System.out.printf("przyjmowanie klienta\n");
        Socket socket = serverSocket.accept();
        System.out.printf("klient przyjęty\n");
        process(socket);
```

[7] Patrz [Lea99], s. 191.

```
        } catch (Exception e) {
          handle(e);
        }
      }
    }

    private void handle(Exception e) {
      if (!(e instanceof SocketException)) {
        e.printStackTrace();
      }
    }

    public void stopProcessing() {
      keepProcessing = false;
      closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
      if (socket == null)
        return;

      try {
        System.out.printf("Serwer: pobieranie komunikatu\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Serwer: komunikat pobrany: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Serwer: wysyłanie odpowiedzi: %s\n", message);
        MessageUtils.sendMessage(socket, "Przetworzone: " + message);
        System.out.printf("Serwer: wysłane\n");
        closeIgnoringException(socket);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    private void closeIgnoringException(Socket socket) {
    if (socket != null)
      try {
        socket.close();
      } catch (IOException ignore) {
      }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
      if (serverSocket != null)
        try {
          serverSocket.close();
        } catch (IOException ignore) {
        }
    }
  }
```

LISTING A4. ClientTest.java

```
  package com.objectmentor.clientserver.nonthreaded;

  import java.io.IOException;
  import java.net.ServerSocket;
  import java.net.Socket;
  import java.net.SocketException;

  import common.MessageUtils;

  public class Server implements Runnable {
```

```
ServerSocket serverSocket;
volatile boolean keepProcessing = true;

public Server(int port, int millisecondsTimeout) throws IOException {
  serverSocket = new ServerSocket(port);
  serverSocket.setSoTimeout(millisecondsTimeout);
}

public void run() {
  System.out.printf("Serwer uruchamia się\n");
  while (keepProcessing) {
    try {
      System.out.printf("przyjmowanie klienta\n");
      Socket socket = serverSocket.accept();
      System.out.printf("klient przyjęty\n");
      process(socket);
    } catch (Exception e) {
      handle(e);
    }
  }
}

private void handle(Exception e) {
  if (!(e instanceof SocketException)) {
    e.printStackTrace();
  }
}

public void stopProcessing() {
  keepProcessing = false;
  closeIgnoringException(serverSocket);
}

void process(Socket socket) {
  if (socket == null)
    return;

  try {
    System.out.printf("Serwer: pobieranie komunikatu\n");
    String message = MessageUtils.getMessage(socket);
    System.out.printf("Serwer: komunikat pobrany: %s\n", message);
    Thread.sleep(1000);
    System.out.printf("Serwer: wysyłanie odpowiedzi: %s\n", message);
    MessageUtils.sendMessage(socket, "Przetworzone: " + message);
    System.out.printf("Serwer: wysłane\n");
    closeIgnoringException(socket);
  } catch (Exception e) {
    e.printStackTrace();
  }
}

private void closeIgnoringException(Socket socket) {
  if (socket != null)
    try {
      socket.close();
    } catch (IOException ignore) {
    }
}

private void closeIgnoringException(ServerSocket serverSocket) {
  if (serverSocket != null)
    try {
      serverSocket.close();
    } catch (IOException ignore) {
```

```
      }
    }
  }
```

LISTING A5. MessageUtils.java

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
  public static void sendMessage(Socket socket, String message)
           throws IOException {
    OutputStream stream = socket.getOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(stream);
    oos.writeUTF(message);
    oos.flush();
  }

  public static String getMessage(Socket socket) throws IOException {
    InputStream stream = socket.getInputStream();
    ObjectInputStream ois = new ObjectInputStream(stream);
    return ois.readUTF();
  }
}
```

Klient-serwer z użyciem wątków

Aby serwer mógł korzystać z wątków, wystarczy zmienić sposób przetwarzania komunikatu (nowe wiersze są — dla ułatwienia — wyróżnione):

```
void process(final Socket socket) {
  if (socket == null)
    return;

  Runnable clientHandler = new Runnable() {
    public void run() {
      try {
        System.out.printf("Serwer: pobieranie komunikatu\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Serwer: komunikat pobrany: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Serwer: wysyłanie odpowiedzi: %s\n", message);
        MessageUtils.sendMessage(socket, "Przetworzone: " + message);
        System.out.printf("Serwer: wysłane\n");
        closeIgnoringException(socket);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  };
  Thread clientConnection = new Thread(clientHandler);
  clientConnection.start();
}
```

org.jfree.date.SerialDate

LISTING B1. SerialDate.java

```
/* ===========================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * ----------------
 * SerialDate.java
 * ----------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
 *
 * Changes (from 11-Oct-2001)
 * --------------------------
 * 11-Oct-2001 : Re-organised the class and moved it to new package
 *               com.jrefinery.date (DG);
```

```
 41  * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
 42  *          class (DG);
 43  * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
 44  *          class is gone (DG);  Changed getPreviousDayOfWeek(),
 45  *          getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
 46  *          bugs (DG);
 47  * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
 48  * 29-May-2002 : Moved the month constants into a separate interface
 49  *          (MonthConstants) (DG);
 50  * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
 51  * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 52  * 13-Mar-2003 : Implemented Serializable (DG);
 53  * 29-May-2003 : Fixed bug in addMonths method (DG);
 54  * 04-Sep-2003 : Implemented Comparable.  Updated the isInRange javadocs (DG);
 55  * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
 56  *
 57  */
 58
 59 package org.jfree.date;
 60
 61 import java.io.Serializable;
 62 import java.text.DateFormatSymbols;
 63 import java.text.SimpleDateFormat;
 64 import java.util.Calendar;
 65 import java.util.GregorianCalendar;
 66
 67 /**
 68  * An abstract class that defines our requirements for manipulating dates,
 69  * without tying down a particular implementation.
 70  * <P>
 71  * Requirement 1 : match at least what Excel does for dates;
 72  * Requirement 2 : class is immutable;
 73  * <P>
 74  * Why not just use java.util.Date?  We will, when it makes sense.  At times,
 75  * java.util.Date can be *too* precise - it represents an instant in time,
 76  * accurate to 1/1000th of a second (with the date itself depending on the
 77  * time-zone).  Sometimes we just want to represent a particular day (e.g. 21
 78  * January 2015) without concerning ourselves about the time of day, or the
 79  * time-zone, or anything else.  That's what we've defined SerialDate for.
 80  * <P>
 81  * You can call getInstance() to get a concrete subclass of SerialDate,
 82  * without worrying about the exact implementation.
 83  *
 84  * @author David Gilbert
 85  */
 86 public abstract class SerialDate implements Comparable,
 87                                              Serializable,
 88                                              MonthConstants {
 89
 90     /** For serialization. */
 91     private static final long serialVersionUID = -2937160404674236 37L;
 92
 93     /** Date format symbols. */
 94     public static final DateFormatSymbols
 95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
 96
 97     /** The serial number for 1 January 1900. */
 98     public static final int SERIAL_LOWER_BOUND = 2;
 99
100     /** The serial number for 31 December 9999. */
101     public static final int SERIAL_UPPER_BOUND = 2958465;
```

```
102
103     /** The lowest year value supported by this date format. */
104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106     /** The highest year value supported by this date format. */
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113      * Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY.
114      */
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118      * Useful constant for Wednesday. Equivalent to
119      * java.util.Calendar.WEDNESDAY.
120      */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124      * Useful constant for Thrusday. Equivalent to java.util.Calendar.THURSDAY.
125      */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132      * Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY.
133      */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
136     /** Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** The number of days in each month in non leap years. */
140     static final int[] LAST_DAY_OF_MONTH =
141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143     /** The number of days in a (non-leap) year up to the end of each month. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** The number of days in a year up to the end of the preceding month. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151     /** The number of days in a leap year up to the end of each month. */
152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155     /**
156      * The number of days in a leap year up to the end of the preceding month.
157      */
158     static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162     /** A useful constant for referring to the first week in a month. */
163     public static final int FIRST_WEEK_IN_MONTH = 1;
```

```
164
165     /** A useful constant for referring to the second week in a month. */
166     public static final int SECOND_WEEK_IN_MONTH = 2;
167
168     /** A useful constant for referring to the third week in a month. */
169     public static final int THIRD_WEEK_IN_MONTH = 3;
170
171     /** A useful constant for referring to the fourth week in a month. */
172     public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174     /** A useful constant for referring to the last week in a month. */
175     public static final int LAST_WEEK_IN_MONTH = 0;
176
177     /** Useful range constant. */
178     public static final int INCLUDE_NONE = 0;
179
180     /** Useful range constant. */
181     public static final int INCLUDE_FIRST = 1;
182
183     /** Useful range constant. */
184     public static final int INCLUDE_SECOND = 2;
185
186     /** Useful range constant. */
187     public static final int INCLUDE_BOTH = 3;
188
189     /**
190      * Useful constant for specifying a day of the week relative to a fixed
191      * date.
192      */
193     public static final int PRECEDING = -1;
194
195     /**
196      * Useful constant for specifying a day of the week relative to a fixed
197      * date.
198      */
199     public static final int NEAREST = 0;
200
201     /**
202      * Useful constant for specifying a day of the week relative to a fixed
203      * date.
204      */
205     public static final int FOLLOWING = 1;
206
207     /** A description for the date. */
208     private String description;
209
210     /**
211      * Default constructor.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Returns <code>true</code> if the supplied integer code represents a
218      * valid day-of-the-week, and <code>false</code> otherwise.
219      *
220      * @param code  the code being checked for validity.
221      *
222      * @return <code>true</code> if the supplied integer code represents a
223      *         valid day-of-the-week, and <code>false</code> otherwise.
224      */
225     public static boolean isValidWeekdayCode(final int code) {
```

```
226
227         switch(code) {
228             case SUNDAY:
229             case MONDAY:
230             case TUESDAY:
231             case WEDNESDAY:
232             case THURSDAY:
233             case FRIDAY:
234             case SATURDAY:
235                 return true;
236             default:
237                 return false;
238         }
239
240     }
241
242     /**
243      * Converts the supplied string to a day of the week.
244      *
245      * @param s  a string representing the day of the week.
246      *
247      * @return <code>-1</code> if the string is not convertable, the day of
248      *      the week otherwise.
249      */
250     public static int stringToWeekdayCode(String s) {
251
252         final String[] shortWeekdayNames
253             = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254         final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256         int result = -1;
257         s = s.trim();
258         for (int i = 0; i < weekDayNames.length; i++) {
259             if (s.equals(shortWeekdayNames[i])) {
260                 result = i;
261                 break;
262             }
263             if (s.equals(weekDayNames[i])) {
264                 result = i;
265                 break;
266             }
267         }
268         return result;
269
270     }
271
272     /**
273      * Returns a string representing the supplied day-of-the-week.
274      * <P>
275      * Need to find a better approach.
276      *
277      * @param weekday  the day of the week.
278      *
279      * @return a string representing the supplied day-of-the-week.
280      */
281     public static String weekdayCodeToString(final int weekday) {
282
283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284         return weekdays[weekday];
285
286     }
287
288     /**
```

```
289       * Returns an array of month names.
290       *
291       * @return an array of month names.
292       */
293      public static String[] getMonths() {
294
295          return getMonths(false);
296
297      }
298
299      /**
300       * Returns an array of month names.
301       *
302       * @param shortened  a flag indicating that shortened month names should
303       *           be returned.
304       *
305       * @return an array of month names.
306       */
307      public static String[] getMonths(final boolean shortened) {
308
309          if (shortened) {
310              return DATE_FORMAT_SYMBOLS.getShortMonths();
311          }
312          else {
313              return DATE_FORMAT_SYMBOLS.getMonths();
314          }
315
316      }
317
318      /**
319       * Returns true if the supplied integer code represents a valid month.
320       *
321       * @param code  the code being checked for validity.
322       *
323       * @return <code>true</code> if the supplied integer code represents a
324       *     valid month.
325       */
326      public static boolean isValidMonthCode(final int code) {
327
328          switch(code) {
329              case JANUARY:
330              case FEBRUARY:
331              case MARCH:
332              case APRIL:
333              case MAY:
334              case JUNE:
335              case JULY:
336              case AUGUST:
337              case SEPTEMBER:
338              case OCTOBER:
339              case NOVEMBER:
340              case DECEMBER:
341                  return true;
342              default:
343                  return false;
344          }
345
346      }
347
348      /**
349       * Returns the quarter for the specified month.
350       *
```

```
351       * @param code  the month code (1-12).
352       *
353       * @return the quarter that the month belongs to.
354       * @throws java.lang.IllegalArgumentException
355       */
356      public static int monthCodeToQuarter(final int code) {
357
358          switch(code) {
359              case JANUARY:
360              case FEBRUARY:
361              case MARCH: return 1;
362              case APRIL:
363              case MAY:
364              case JUNE: return 2;
365              case JULY:
366              case AUGUST:
367              case SEPTEMBER: return 3;
368              case OCTOBER:
369              case NOVEMBER:
370              case DECEMBER: return 4;
371              default: throw new IllegalArgumentException(
372                  "SerialDate.monthCodeToQuarter: invalid month code.");
373          }
374
375      }
376
377      /**
378       * Returns a string representing the supplied month.
379       * <P>
380       * The string returned is the long form of the month name taken from the
381       * default locale.
382       *
383       * @param month  the month.
384       *
385       * @return a string representing the supplied month.
386       */
387      public static String monthCodeToString(final int month) {
388
389          return monthCodeToString(month, false);
390
391      }
392
393      /**
394       * Returns a string representing the supplied month.
395       * <P>
396       * The string returned is the long or short form of the month name taken
397       * from the default locale.
398       *
399       * @param month  the month.
400       * @param shortened  if <code>true</code> return the abbreviation of the
401       *              month.
402       *
403       * @return a string representing the supplied month.
404       * @throws java.lang.IllegalArgumentException
405       */
406      public static String monthCodeToString(final int month,
407                                             final boolean shortened) {
408
409          // check arguments...
410          if (!isValidMonthCode(month)) {
411              throw new IllegalArgumentException(
412                  "SerialDate.monthCodeToString: month outside valid range.");
```

```
413         }
414
415         final String[] months;
416
417         if (shortened) {
418             months = DATE_FORMAT_SYMBOLS.getShortMonths();
419         }
420         else {
421             months = DATE_FORMAT_SYMBOLS.getMonths();
422         }
423
424         return months[month - 1];
425
426     }
427
428     /**
429      * Converts a string to a month code.
430      * <P>
431      * This method will return one of the constants JANUARY, FEBRUARY, ...,
432      * DECEMBER that corresponds to the string.  If the string is not
433      * recognised, this method returns -1.
434      *
435      * @param s  the string to parse.
436      *
437      * @return <code>-1</code> if the string is not parseable, the month of the
438      *     year otherwise.
439      */
440     public static int stringToMonthCode(String s) {
441
442         final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
443         final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
444
445         int result = -1;
446         s = s.trim();
447
448         // first try parsing the string as an integer (1-12)...
449         try {
450             result = Integer.parseInt(s);
451         }
452         catch (NumberFormatException e) {
453             // suppress
454         }
455
456         // now search through the month names...
457         if ((result < 1) || (result > 12)) {
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equals(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equals(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
469
470         return result;
471
472     }
473
474     /**
475      * Returns true if the supplied integer code represents a valid
```

```
476       * week-in-the-month, and false otherwise.
477       *
478       * @param code  the code being checked for validity.
479       * @return <code>true</code> if the supplied integer code represents a
480       *      valid week-in-the-month.
481       */
482      public static boolean isValidWeekInMonthCode(final int code) {
483
484          switch(code) {
485              case FIRST_WEEK_IN_MONTH:
486              case SECOND_WEEK_IN_MONTH:
487              case THIRD_WEEK_IN_MONTH:
488              case FOURTH_WEEK_IN_MONTH:
489              case LAST_WEEK_IN_MONTH: return true;
490              default: return false;
491          }
492
493      }
494
495      /**
496       * Determines whether or not the specified year is a leap year.
497       *
498       * @param yyyy  the year (in the range 1900 to 9999).
499       *
500       * @return <code>true</code> if the specified year is a leap year.
501       */
502      public static boolean isLeapYear(final int yyyy) {
503
504          if ((yyyy % 4) != 0) {
505              return false;
506          }
507          else if ((yyyy % 400) == 0) {
508              return true;
509          }
510          else if ((yyyy % 100) == 0) {
511              return false;
512          }
513          else {
514              return true;
515          }
516
517      }
518
519      /**
520       * Returns the number of leap years from 1900 to the specified year
521       * INCLUSIVE.
522       * <P>
523       * Note that 1900 is not a leap year.
524       *
525       * @param yyyy  the year (in the range 1900 to 9999).
526       *
527       * @return the number of leap years from 1900 to the specified year.
528       */
529      public static int leapYearCount(final int yyyy) {
530
531          final int leap4 = (yyyy - 1896) / 4;
532          final int leap100 = (yyyy - 1800) / 100;
533          final int leap400 = (yyyy - 1600) / 400;
534          return leap4 - leap100 + leap400;
535
536      }
537
```

```
538     /**
539      * Returns the number of the last day of the month, taking into account
540      * leap years.
541      *
542      * @param month  the month.
543      * @param yyyy  the year (in the range 1900 to 9999).
544      *
545      * @return the number of the last day of the month.
546      */
547     public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
551             return result;
552         }
553         else if (isLeapYear(yyyy)) {
554             return result + 1;
555         }
556         else {
557             return result;
558         }
559
560     }
561
562     /**
563      * Creates a new date by adding the specified number of days to the base
564      * date.
565      *
566      * @param days  the number of days to add (can be negative).
567      * @param base  the base date.
568      *
569      * @return a new date.
570      */
571     public static SerialDate addDays(final int days, final SerialDate base) {
572
573         final int serialDayNumber = base.toSerial() + days;
574         return SerialDate.createInstance(serialDayNumber);
575
576     }
577
578     /**
579      * Creates a new date by adding the specified number of months to the base
580      * date.
581      * <P>
582      * If the base date is close to the end of the month, the day on the result
583      * may be adjusted slightly:  31 May + 1 month = 30 June.
584      *
585      * @param months  the number of months to add (can be negative).
586      * @param base  the base date.
587      *
588      * @return a new date.
589      */
590     public static SerialDate addMonths(final int months,
591                                        final SerialDate base) {
592
593         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594                        / 12;
595         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
596                        % 12 + 1;
597         final int dd = Math.min(
598             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
599         );
```

```
600         return SerialDate.createInstance(dd, mm, yy);
601
602     }
603
604     /**
605      * Creates a new date by adding the specified number of years to the base
606      * date.
607      *
608      * @param years  the number of years to add (can be negative).
609      * @param base  the base date.
610      *
611      * @return A new date.
612      */
613     public static SerialDate addYears(final int years, final SerialDate base) {
614
615         final int baseY = base.getYYYY();
616         final int baseM = base.getMonth();
617         final int baseD = base.getDayOfMonth();
618
619         final int targetY = baseY + years;
620         final int targetD = Math.min(
621             baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622         );
623
624         return SerialDate.createInstance(targetD, baseM, targetY);
625
626     }
627
628     /**
629      * Returns the latest date that falls on the specified day-of-the-week and
630      * is BEFORE the base date.
631      *
632      * @param targetWeekday  a code for the target day-of-the-week.
633      * @param base  the base date.
634      *
635      * @return the latest date that falls on the specified day-of-the-week and
636      *     is BEFORE the base date.
637      */
638     public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                                    final SerialDate base) {
640
641         // check arguments...
642         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643             throw new IllegalArgumentException(
644                 "Invalid day-of-the-week code."
645             );
646         }
647
648         // find the date...
649         final int adjust;
650         final int baseDOW = base.getDayOfWeek();
651         if (baseDOW > targetWeekday) {
652             adjust = Math.min(0, targetWeekday - baseDOW);
653         }
654         else {
655             adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656         }
657
658         return SerialDate.addDays(adjust, base);
659
660     }
661
```

```
662     /**
663      * Returns the earliest date that falls on the specified day-of-the-week
664      * and is AFTER the base date.
665      *
666      * @param targetWeekday  a code for the target day-of-the-week.
667      * @param base  the base date.
668      *
669      * @return the earliest date that falls on the specified day-of-the-week
670      *     and is AFTER the base date.
671      */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                                     final SerialDate base) {
674
675         // check arguments...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code."
679             );
680         }
681
682         // find the date...
683         final int adjust;
684         final int baseDOW = base.getDayOfWeek();
685         if (baseDOW > targetWeekday) {
686             adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687         }
688         else {
689             adjust = Math.max(0, targetWeekday - baseDOW);
690         }
691
692         return SerialDate.addDays(adjust, base);
693     }
694
695     /**
696      * Returns the date that falls on the specified day-of-the-week and is
697      * CLOSEST to the base date.
698      *
699      * @param targetDOW  a code for the target day-of-the-week.
700      * @param base  the base date.
701      *
702      * @return the date that falls on the specified day-of-the-week and is
703      *     CLOSEST to the base date.
704      */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                                   final SerialDate base) {
707
708         // check arguments...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code."
712             );
713         }
714
715         // find the date...
716         final int baseDOW = base.getDayOfWeek();
717         int adjust = -Math.abs(targetDOW - baseDOW);
718         if (adjust >= 4) {
719             adjust = 7 - adjust;
720         }
721         if (adjust <= -4) {
722             adjust = 7 + adjust;
723         }
```

```
724         return SerialDate.addDays(adjust, base);
725
726     }
727
728     /**
729      * Rolls the date forward to the last day of the month.
730      *
731      * @param base  the base date.
732      *
733      * @return a new serial date.
734      */
735     public SerialDate getEndOfCurrentMonth(final SerialDate base) {
736         final int last = SerialDate.lastDayOfMonth(
737             base.getMonth(), base.getYYYY()
738         );
739         return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
740     }
741
742     /**
743      * Returns a string corresponding to the week-in-the-month code.
744      * <P>
745      * Need to find a better approach.
746      *
747      * @param count  an integer code representing the week-in-the-month.
748      *
749      * @return a string corresponding to the week-in-the-month code.
750      */
751     public static String weekInMonthToString(final int count) {
752
753         switch (count) {
754             case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
755             case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
756             case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
757             case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
758             case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
759             default :
760                 return "SerialDate.weekInMonthToString(): invalid code.";
761         }
762
763     }
764
765     /**
766      * Returns a string representing the supplied 'relative'.
767      * <P>
768      * Need to find a better approach.
769      *
770      * @param relative  a constant representing the 'relative'.
771      *
772      * @return a string representing the supplied 'relative'.
773      */
774     public static String relativeToString(final int relative) {
775
776         switch (relative) {
777             case SerialDate.PRECEDING : return "Preceding";
778             case SerialDate.NEAREST : return "Nearest";
779             case SerialDate.FOLLOWING : return "Following";
780             default : return "ERROR : Relative To String";
781         }
782
783     }
784
785     /**
```

```
786       * Factory method that returns an instance of some concrete subclass of
787       * {@link SerialDate}.
788       *
789       * @param day  the day (1-31).
790       * @param month  the month (1-12).
791       * @param yyyy  the year (in the range 1900 to 9999).
792       *
793       * @return An instance of {@link SerialDate}.
794       */
795      public static SerialDate createInstance(final int day, final int month,
796                                               final int yyyy) {
797          return new SpreadsheetDate(day, month, yyyy);
798      }
799
800      /**
801       * Factory method that returns an instance of some concrete subclass of
802       * {@link SerialDate}.
803       *
804       * @param serial  the serial number for the day (1 January 1900 = 2).
805       *
806       * @return a instance of SerialDate.
807       */
808      public static SerialDate createInstance(final int serial) {
809          return new SpreadsheetDate(serial);
810      }
811
812      /**
813       * Factory method that returns an instance of a subclass of SerialDate.
814       *
815       * @param date  A Java date object.
816       *
817       * @return a instance of SerialDate.
818       */
819      public static SerialDate createInstance(final java.util.Date date) {
820
821          final GregorianCalendar calendar = new GregorianCalendar();
822          calendar.setTime(date);
823          return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                     calendar.get(Calendar.MONTH) + 1,
825                                     calendar.get(Calendar.YEAR));
826
827      }
828
829      /**
830       * Returns the serial number for the date, where 1 January 1900 = 2 (this
831       * corresponds, almost, to the numbering system used in Microsoft Excel for
832       * Windows and Lotus 1-2-3).
833       *
834       * @return the serial number for the date.
835       */
836      public abstract int toSerial();
837
838      /**
839       * Returns a java.util.Date.  Since java.util.Date has more precision than
840       * SerialDate, we need to define a convention for the 'time of day'.
841       *
842       * @return this as <code>java.util.Date</code>.
843       */
844      public abstract java.util.Date toDate();
845
846      /**
```

```
847       * Returns a description of the date.
848       *
849       * @return a description of the date.
850       */
851      public String getDescription() {
852          return this.description;
853      }
854
855      /**
856       * Sets the description for the date.
857       *
858       * @param description  the new description for the date.
859       */
860      public void setDescription(final String description) {
861          this.description = description;
862      }
863
864      /**
865       * Converts the date to a string.
866       *
867       * @return  a string representation of the date.
868       */
869      public String toString() {
870          return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871                                 + "-" + getYYYY();
872      }
873
874      /**
875       * Returns the year (assume a valid range of 1900 to 9999).
876       *
877       * @return the year.
878       */
879      public abstract int getYYYY();
880
881      /**
882       * Returns the month (January = 1, February = 2, March = 3).
883       *
884       * @return the month of the year.
885       */
886      public abstract int getMonth();
887
888      /**
889       * Returns the day of the month.
890       *
891       * @return the day of the month.
892       */
893      public abstract int getDayOfMonth();
894
895      /**
896       * Returns the day of the week.
897       *
898       * @return the day of the week.
899       */
900      public abstract int getDayOfWeek();
901
902      /**
903       * Returns the difference (in days) between this date and the specified
904       * 'other' date.
905       * <P>
906       * The result is positive if this date is after the 'other' date and
907       * negative if it is before the 'other' date.
```

```
908       *
909       * @param other  the date being compared to.
910       *
911       * @return the difference between this and the other date.
912       */
913      public abstract int compare(SerialDate other);
914
915      /**
916       * Returns true if this SerialDate represents the same date as the
917       * specified SerialDate.
918       *
919       * @param other  the date being compared to.
920       *
921       * @return <code>true</code> if this SerialDate represents the same date as
922       *         the specified SerialDate.
923       */
924      public abstract boolean isOn(SerialDate other);
925
926      /**
927       * Returns true if this SerialDate represents an earlier date compared to
928       * the specified SerialDate.
929       *
930       * @param other  The date being compared to.
931       *
932       * @return <code>true</code> if this SerialDate represents an earlier date
933       *         compared to the specified SerialDate.
934       */
935      public abstract boolean isBefore(SerialDate other);
936
937      /**
938       * Returns true if this SerialDate represents the same date as the
939       * specified SerialDate.
940       *
941       * @param other  the date being compared to.
942       *
943       * @return <code>true<code> if this SerialDate represents the same date
944       *         as the specified SerialDate.
945       */
946      public abstract boolean isOnOrBefore(SerialDate other);
947
948      /**
949       * Returns true if this SerialDate represents the same date as the
950       * specified SerialDate.
951       *
952       * @param other  the date being compared to.
953       *
954       * @return <code>true</code> if this SerialDate represents the same date
955       *         as the specified SerialDate.
956       */
957      public abstract boolean isAfter(SerialDate other);
958
959      /**
960       * Returns true if this SerialDate represents the same date as the
961       * specified SerialDate.
962       *
963       * @param other  the date being compared to.
964       *
965       * @return <code>true</code> if this SerialDate represents the same date
966       *         as the specified SerialDate.
967       */
968      public abstract boolean isOnOrAfter(SerialDate other);
```

```
969
970     /**
971      * Returns <code>true</code> if this {@link SerialDate} is within the
972      * specified range (INCLUSIVE).  The date order of d1 and d2 is not
973      * important.
974      *
975      * @param d1  a boundary date for the range.
976      * @param d2  the other boundary date for the range.
977      *
978      * @return A boolean.
979      */
980     public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982     /**
983      * Returns <code>true</code> if this {@link SerialDate} is within the
984      * specified range (caller specifies whether or not the end-points are
985      * included).  The date order of d1 and d2 is not important.
986      *
987      * @param d1  a boundary date for the range.
988      * @param d2  the other boundary date for the range.
989      * @param include  a code that controls whether or not the start and end
990      *                 dates are included in the range.
991      *
992      * @return A boolean.
993      */
994     public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995                                       int include);
996
997     /**
998      * Returns the latest date that falls on the specified day-of-the-week and
999      * is BEFORE this date.
1000     *
1001     * @param targetDOW  a code for the target day-of-the-week.
1002     *
1003     * @return the latest date that falls on the specified day-of-the-week and
1004     *      is BEFORE this date.
1005     */
1006    public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007        return getPreviousDayOfWeek(targetDOW, this);
1008    }
1009
1010    /**
1011     * Returns the earliest date that falls on the specified day-of-the-week
1012     * and is AFTER this date.
1013     *
1014     * @param targetDOW  a code for the target day-of-the-week.
1015     *
1016     * @return the earliest date that falls on the specified day-of-the-week
1017     *      and is AFTER this date.
1018     */
1019    public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020        return getFollowingDayOfWeek(targetDOW, this);
1021    }
1022
1023    /**
1024     * Returns the nearest date that falls on the specified day-of-the-week.
1025     *
1026     * @param targetDOW  a code for the target day-of-the-week.
1027     *
1028     * @return the nearest date that falls on the specified day-of-the-week.
1029     */
```

```
1030     public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031         return getNearestDayOfWeek(targetDOW, this);
1032     }
1033
1034 }
```

LISTING B2. SerialDateTest.java

```
 1 /* ===========================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ===========================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Project Info:  http://www.jfree.org/jcommon/index.html
 8  *
 9  * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * --------------------
28  * SerialDateTests.java
29  * --------------------
30  * (C) Copyright 2001-2005, by Object Refinery Limited.
31  *
32  * Original Author:  David Gilbert (for Object Refinery Limited);
33  * Contributor(s):   -;
34  *
35  * $Id: SerialDateTests.java,v 1.5 2005/10/18 13:15:28 mungady Exp $
36  *
37  * Changes
38  * -------
39  * 15-Nov-2001 : Version 1 (DG);
40  * 25-Jun-2002 : Removed unnecessary import (DG);
41  * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
42  * 13-Mar-2003 : Added serialization test (DG);
43  * 05-Jan-2005 : Added test for bug report 1096282 (DG);
44  *
45  */
46
47 package org.jfree.date.junit;
48
49 import java.io.ByteArrayInputStream;
50 import java.io.ByteArrayOutputStream;
51 import java.io.ObjectInput;
52 import java.io.ObjectInputStream;
53 import java.io.ObjectOutput;
```

```
54 import java.io.ObjectOutputStream;
55
56 import junit.framework.Test;
57 import junit.framework.TestCase;
58 import junit.framework.TestSuite;
59
60 import org.jfree.date.MonthConstants;
61 import org.jfree.date.SerialDate;
62
63 /**
64  * Some JUnit tests for the {@link SerialDate} class.
65  */
66 public class SerialDateTests extends TestCase {
67
68     /** Date representing November 9. */
69     private SerialDate nov9Y2001;
70
71     /**
72      * Creates a new test case.
73      *
74      * @param name  the name.
75      */
76     public SerialDateTests(final String name) {
77         super(name);
78     }
79
80     /**
81      * Returns a test suite for the JUnit test runner.
82      *
83      * @return The test suite.
84      */
85     public static Test suite() {
86         return new TestSuite(SerialDateTests.class);
87     }
88
89     /**
90      * Problem set up.
91      */
92     protected void setUp() {
93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
94     }
95
96     /**
97      * 9 Nov 2001 plus two months should be 9 Jan 2002.
98      */
99     public void testAddMonthsTo9Nov2001() {
100        final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101        final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102        assertEquals(answer, jan9Y2002);
103    }
104
105    /**
106     * A test case for a reported bug, now fixed.
107     */
108    public void testAddMonthsTo5Oct2003() {
109       final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110        final SerialDate d2 = SerialDate.addMonths(2, d1);
111        assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112    }
113
114    /**
115     * A test case for a reported bug, now fixed.
```

```
116      */
117     public void testAddMonthsTo1Jan2003() {
118         final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119         final SerialDate d2 = SerialDate.addMonths(0, d1);
120         assertEquals(d2, d1);
121     }
122
123     /**
124      * Monday preceding Friday 9 November 2001 should be 5 November.
125      */
126     public void testMondayPrecedingFriday9Nov2001() {
127         SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128             SerialDate.MONDAY, this.nov9Y2001
129         );
130         assertEquals(5, mondayBefore.getDayOfMonth());
131     }
132
133     /**
134      * Monday following Friday 9 November 2001 should be 12 November.
135      */
136     public void testMondayFollowingFriday9Nov2001() {
137         SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138             SerialDate.MONDAY, this.nov9Y2001
139         );
140         assertEquals(12, mondayAfter.getDayOfMonth());
141     }
142
143     /**
144      * Monday nearest Friday 9 November 2001 should be 12 November.
145      */
146     public void testMondayNearestFriday9Nov2001() {
147         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148             SerialDate.MONDAY, this.nov9Y2001
149         );
150         assertEquals(12, mondayNearest.getDayOfMonth());
151     }
152
153     /**
154      * The Monday nearest to 22nd January 1970 falls on the 19th.
155      */
156     public void testMondayNearest22Jan1970() {
157         SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY,
             ↪1970);
158         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(SerialDate.MONDAY,
             ↪jan22Y1970);
159         assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161
162     /**
163      * Problem that the conversion of days to strings returns the right result.  Actually, this
164      * result depends on the Locale so this test needs to be modified.
165      */
166     public void testWeekdayCodeToString() {
167
168         final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169         assertEquals("Saturday", test);
170
171     }
172
173     /**
174      * Test the conversion of a string to a weekday.  Note that this test will fail if the
175      * default locale doesn't use English weekday names...devise a better test!
```

```
176      */
177     public void testStringToWeekday() {
178
179         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180         assertEquals(SerialDate.WEDNESDAY, weekday);
181
182         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183         assertEquals(SerialDate.WEDNESDAY, weekday);
184
185         weekday = SerialDate.stringToWeekdayCode("Wed");
186         assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     }
189
190     /**
191      * Test the conversion of a string to a month.  Note that this test will fail if the default
192      * locale doesn't use English month names...devise a better test!
193      */
194     public void testStringToMonthCode() {
195
196         int m = SerialDate.stringToMonthCode("January");
197         assertEquals(MonthConstants.JANUARY, m);
198
199         m = SerialDate.stringToMonthCode(" January ");
200         assertEquals(MonthConstants.JANUARY, m);
201
202         m = SerialDate.stringToMonthCode("Jan");
203         assertEquals(MonthConstants.JANUARY, m);
204
205     }
206
207     /**
208      * Tests the conversion of a month code to a string.
209      */
210     public void testMonthCodeToStringCode() {
211
212         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213         assertEquals("December", test);
214
215     }
216
217     /**
218      * 1900 is not a leap year.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * 2000 is a leap year.
226      */
227     public void testIsLeapYear2000() {
228         assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * The number of leap years from 1900 up-to-and-including 1899 is 0.
233      */
234     public void testLeapYearCount1899() {
235         assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
```

```
238     /**
239      * The number of leap years from 1900 up-to-and-including 1903 is 0.
240      */
241     public void testLeapYearCount1903() {
242         assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * The number of leap years from 1900 up-to-and-including 1904 is 1.
247      */
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253      * The number of leap years from 1900 up-to-and-including 1999 is 24.
254      */
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * The number of leap years from 1900 up-to-and-including 2000 is 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Serialize an instance, restore it, and check for equality.
268      */
269     public void testSerialization() {
270
271         SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272         SerialDate d2 = null;
273
274         try {
275             ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276             ObjectOutput out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInput in = new ObjectInputStream(new ByteArrayInputStream
                ↪(buffer.toByteArray()));
281             d2 = (SerialDate) in.readObject();
282             in.close();
283         }
284         catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288
289     }
290
291     /**
292      * A test for bug report 1096282 (now fixed).
293      */
294     public void test1096282() {
295         SerialDate d = SerialDate.createInstance(29, 2, 2004);
296         d = SerialDate.addYears(1, d);
297         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298         assertTrue(d.isOn(expected));
```

```
299     }
300
301     /**
302      * Miscellaneous tests for the addMonths() method.
303      */
304     public void testAddMonths() {
305         SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
306
307         SerialDate d2 = SerialDate.addMonths(1, d1);
308         assertEquals(30, d2.getDayOfMonth());
309         assertEquals(6, d2.getMonth());
310         assertEquals(2004, d2.getYYYY());
311
312         SerialDate d3 = SerialDate.addMonths(2, d1);
313         assertEquals(31, d3.getDayOfMonth());
314         assertEquals(7, d3.getMonth());
315         assertEquals(2004, d3.getYYYY());
316
317         SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318         assertEquals(30, d4.getDayOfMonth());
319         assertEquals(7, d4.getMonth());
320         assertEquals(2004, d4.getYYYY());
321     }
322 }
```

LISTING B3. MonthConstants.java

```
 1 /* ===========================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ===========================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Project Info:  http://www.jfree.org/jcommon/index.html
 8  *
 9  * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * ------------------
28  * MonthConstants.java
29  * ------------------
30  * (C) Copyright 2002, 2003, by Object Refinery Limited.
31  *
32  * Original Author:  David Gilbert (for Object Refinery Limited);
33  * Contributor(s):   -;
```

```
 *
 * $Id: MonthConstants.java,v 1.3 2005/10/18 13:15:08 mungady Exp $
 *
 * Changes
 * -------
 * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
 *
 */

package org.jfree.date;

/**
 * Useful constants for months.  Note that these are NOT equivalent to the
 * constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
 * <P>
 * Used by the SerialDate and RegularTimePeriod classes.
 *
 * @author David Gilbert
 */
public interface MonthConstants {

    /** Constant for January. */
    public static final int JANUARY = 1;

    /** Constant for February. */
    public static final int FEBRUARY = 2;

    /** Constant for March. */
    public static final int MARCH = 3;

    /** Constant for April. */
    public static final int APRIL = 4;

    /** Constant for May. */
    public static final int MAY = 5;

    /** Constant for June. */
    public static final int JUNE = 6;

    /** Constant for July. */
    public static final int JULY = 7;

    /** Constant for August. */
    public static final int AUGUST = 8;

    /** Constant for September. */
    public static final int SEPTEMBER = 9;

    /** Constant for October. */
    public static final int OCTOBER = 10;

    /** Constant for November. */
    public static final int NOVEMBER = 11;

    /** Constant for December. */
    public static final int DECEMBER = 12;

}
```

LISTING B4. BobsSerialDateTest.java

```
 1 package org.jfree.date.junit;
 2
 3 import junit.framework.TestCase;
 4 import org.jfree.date.*;
 5 import static org.jfree.date.SerialDate.*;
 6
 7 import java.util.*;
 8
 9 public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12     for (int day = 1; day <= 7; day++)
13       assertTrue(isValidWeekdayCode(day));
14     assertFalse(isValidWeekdayCode(0));
15     assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20     assertEquals(-1, stringToWeekdayCode("Hello"));
21     assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22     assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo assertEquals(MONDAY,stringToWeekdayCode("monday"));
24 // assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));
25 // assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27     assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28     assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 // assertEquals(TUESDAY,stringToWeekdayCode("tuesday"));
30 // assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY"));
31 // assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 // assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36 // assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday"));
37 // assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY"));
38 // assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42 // assertEquals(THURSDAY,stringToWeekdayCode("thursday"));
43 // assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY"));
44 // assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 // assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 // assertEquals(FRIDAY,stringToWeekdayCode("friday"));
50 // assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY"));
51 // assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
55 // assertEquals(SATURDAY,stringToWeekdayCode("saturday"));
56 // assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY"));
57 // assertEquals(SATURDAY, stringToWeekdayCode("sat"));
58
59     assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
60     assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
61 // assertEquals(SUNDAY,stringToWeekdayCode("sunday"));
```

```
62 // assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY"));
63 // assertEquals(SUNDAY, stringToWeekdayCode("sun"));
64   }
65
66   public void testWeekdayCodeToString() throws Exception {
67     assertEquals("Sunday", weekdayCodeToString(SUNDAY));
68     assertEquals("Monday", weekdayCodeToString(MONDAY));
69     assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
70     assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
71     assertEquals("Thursday", weekdayCodeToString(THURSDAY));
72     assertEquals("Friday", weekdayCodeToString(FRIDAY));
73     assertEquals("Saturday", weekdayCodeToString(SATURDAY));
74   }
75
76   public void testIsValidMonthCode() throws Exception {
77     for (int i = 1; i <= 12; i++)
78       assertTrue(isValidMonthCode(i));
79     assertFalse(isValidMonthCode(0));
80     assertFalse(isValidMonthCode(13));
81   }
82
83   public void testMonthToQuarter() throws Exception {
84     assertEquals(1, monthCodeToQuarter(JANUARY));
85     assertEquals(1, monthCodeToQuarter(FEBRUARY));
86     assertEquals(1, monthCodeToQuarter(MARCH));
87     assertEquals(2, monthCodeToQuarter(APRIL));
88     assertEquals(2, monthCodeToQuarter(MAY));
89     assertEquals(2, monthCodeToQuarter(JUNE));
90     assertEquals(3, monthCodeToQuarter(JULY));
91     assertEquals(3, monthCodeToQuarter(AUGUST));
92     assertEquals(3, monthCodeToQuarter(SEPTEMBER));
93     assertEquals(4, monthCodeToQuarter(OCTOBER));
94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
95     assertEquals(4, monthCodeToQuarter(DECEMBER));
96
97     try {
98       monthCodeToQuarter(-1);
99       fail("Nieprawidłowy kod miesiąca powinien wygenerować wyjątek");
100    } catch (IllegalArgumentException e) {
101    }
102  }
103
104  public void testMonthCodeToString() throws Exception {
105    assertEquals("January", monthCodeToString(JANUARY));
106    assertEquals("February", monthCodeToString(FEBRUARY));
107    assertEquals("March", monthCodeToString(MARCH));
108    assertEquals("April", monthCodeToString(APRIL));
109    assertEquals("May", monthCodeToString(MAY));
110    assertEquals("June", monthCodeToString(JUNE));
111    assertEquals("July", monthCodeToString(JULY));
112    assertEquals("August", monthCodeToString(AUGUST));
113    assertEquals("September", monthCodeToString(SEPTEMBER));
114    assertEquals("October", monthCodeToString(OCTOBER));
115    assertEquals("November", monthCodeToString(NOVEMBER));
116    assertEquals("December", monthCodeToString(DECEMBER));
117
118    assertEquals("Jan", monthCodeToString(JANUARY, true));
119    assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120    assertEquals("Mar", monthCodeToString(MARCH, true));
121    assertEquals("Apr", monthCodeToString(APRIL, true));
122    assertEquals("May", monthCodeToString(MAY, true));
123    assertEquals("Jun", monthCodeToString(JUNE, true));
124    assertEquals("Jul", monthCodeToString(JULY, true));
125    assertEquals("Aug", monthCodeToString(AUGUST, true));
```

```
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
128     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129     assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132       monthCodeToString(-1);
133       fail("Nieprawidłowy kod miesiąca powinien wygenerować wyjątek");
134     } catch (IllegalArgumentException e) {
135     }
136
137   }
138
139   public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY,stringToMonthCode("1"));
141     assertEquals(FEBRUARY,stringToMonthCode("2"));
142     assertEquals(MARCH,stringToMonthCode("3"));
143     assertEquals(APRIL,stringToMonthCode("4"));
144     assertEquals(MAY,stringToMonthCode("5"));
145     assertEquals(JUNE,stringToMonthCode("6"));
146     assertEquals(JULY,stringToMonthCode("7"));
147     assertEquals(AUGUST,stringToMonthCode("8"));
148     assertEquals(SEPTEMBER,stringToMonthCode("9"));
149     assertEquals(OCTOBER,stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER,stringToMonthCode("12"));
152
153 //todo assertEquals(-1, stringToMonthCode("0"));
154 // assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1,stringToMonthCode("Cześć"));
157
158     for (int m = 1; m <= 12; m++) {
159       assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160       assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161     }
162
163 // assertEquals(1,stringToMonthCode("jan"));
164 // assertEquals(2,stringToMonthCode("feb"));
165 // assertEquals(3,stringToMonthCode("mar"));
166 // assertEquals(4,stringToMonthCode("apr"));
167 // assertEquals(5,stringToMonthCode("may"));
168 // assertEquals(6,stringToMonthCode("jun"));
169 // assertEquals(7,stringToMonthCode("jul"));
170 // assertEquals(8,stringToMonthCode("aug"));
171 // assertEquals(9,stringToMonthCode("sep"));
172 // assertEquals(10,stringToMonthCode("oct"));
173 // assertEquals(11,stringToMonthCode("nov"));
174 // assertEquals(12,stringToMonthCode("dec"));
175
176 // assertEquals(1,stringToMonthCode("JAN"));
177 // assertEquals(2,stringToMonthCode("FEB"));
178 // assertEquals(3,stringToMonthCode("MAR"));
179 // assertEquals(4,stringToMonthCode("APR"));
180 // assertEquals(5,stringToMonthCode("MAY"));
181 // assertEquals(6,stringToMonthCode("JUN"));
182 // assertEquals(7,stringToMonthCode("JUL"));
183 // assertEquals(8,stringToMonthCode("AUG"));
184 // assertEquals(9,stringToMonthCode("SEP"));
185 // assertEquals(10,stringToMonthCode("OCT"));
186 // assertEquals(11,stringToMonthCode("NOV"));
187 // assertEquals(12,stringToMonthCode("DEC"));
```

```
188
189 // assertEquals(1,stringToMonthCode("january"));
190 // assertEquals(2,stringToMonthCode("february"));
191 // assertEquals(3,stringToMonthCode("march"));
192 // assertEquals(4,stringToMonthCode("april"));
193 // assertEquals(5,stringToMonthCode("may"));
194 // assertEquals(6,stringToMonthCode("june"));
195 // assertEquals(7,stringToMonthCode("july"));
196 // assertEquals(8,stringToMonthCode("august"));
197 // assertEquals(9,stringToMonthCode("september"));
198 // assertEquals(10,stringToMonthCode("october"));
199 // assertEquals(11,stringToMonthCode("november"));
200 // assertEquals(12,stringToMonthCode("december"));
201
202 // assertEquals(1,stringToMonthCode("JANUARY"));
203 // assertEquals(2,stringToMonthCode("FEBRUARY"));
204 // assertEquals(3,stringToMonthCode("MAR"));
205 // assertEquals(4,stringToMonthCode("APRIL"));
206 // assertEquals(5,stringToMonthCode("MAY"));
207 // assertEquals(6,stringToMonthCode("JUNE"));
208 // assertEquals(7,stringToMonthCode("JULY"));
209 // assertEquals(8,stringToMonthCode("AUGUST"));
210 // assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 // assertEquals(10,stringToMonthCode("OCTOBER"));
212 // assertEquals(11,stringToMonthCode("NOVEMBER"));
213 // assertEquals(12,stringToMonthCode("DECEMBER"));
214   }
215
216   public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218       assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221   }
222
223   public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
```

```
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year) {return new
      ↪SpreadsheetDate(day, month, year);}
281
282   public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
295   }
296
297   public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302   }
303
304   public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH,
        ↪2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH,
        ↪2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH,
        ↪2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5,
        ↪JANUARY, 2005)));
```

```
309
310     try {
311       getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312       fail("Nieprawidłowy kod tygodnia powinien wygenerować wyjątek");
313     } catch (IllegalArgumentException e) {
314     }
315   }
316
317   public void testGetFollowingDayOfWeek() throws Exception {
318 // assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER,
       ↪2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY,
       ↪2004)));
321
322     try {
323       getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324       fail("Nieprawidłowy kod tygodnia powinien wygenerować wyjątek");
325     } catch (IllegalArgumentException e) {
326     }
327   }
328
329   public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 // assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
```

```
370 // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386     try {
387       getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388       fail("Nieprawidłowy kod tygodnia powinien wygenerować wyjątek");
389     } catch (IllegalArgumentException e) {
390     }
391   }
392
393   public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408   }
409
410   public void testWeekInMonthToString() throws Exception {
411     assertEquals("First",weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second",weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third",weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth",weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last",weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo try {
418 // weekInMonthToString(-1);
419 // fail("Nieprawidłowy kod tygodnia powinien wygenerować wyjątek");
420 // } catch (IllegalArgumentException e) {
421 // }
422   }
423
424   public void testRelativeToString() throws Exception {
425     assertEquals("Preceding",relativeToString(PRECEDING));
426     assertEquals("Nearest",relativeToString(NEAREST));
427     assertEquals("Following",relativeToString(FOLLOWING));
428
429 //todo try {
430 // relativeToString(-1000);
431 // fail("Nieprawidłowy kod względny powinien wygenerować wyjątek");
432 // } catch (IllegalArgumentException e) {
```

```
433 //}
434   }
435
436   public void testCreateInstanceFromDDMMYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1,date.getDayOfMonth());
439     assertEquals(JANUARY,date.getMonth());
440     assertEquals(1900,date.getYYYY());
441     assertEquals(2,date.toSerial());
442   }
443
444   public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900),createInstance(2));
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));
447   }
448
449   public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),createInstance(new GregorianCalendar
        ↪(1900,0,1).getTime()));
451     assertEquals(d(1, JANUARY, 2006),createInstance(new GregorianCalendar
        ↪(2006,0,1).getTime()));
452   }
453
454   public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456   }
457 }
```

LISTING B5. SpreadsheetDate.java

```
 1 /* ===========================================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Project Info:  http://www.jfree.org/jcommon/index.html
 8  *
 9  * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * ---------------------
28  * SpreadsheetDate.java
29  * ---------------------
30  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
31  *
32  * Original Author:  David Gilbert (for Object Refinery Limited);
```

```
33   * Contributor(s):   -;
34   *
35   * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36   *
37   * Changes
38   * -------
39   * 11-Oct-2001 : Version 1 (DG);
40   * 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
41   * 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
42   *               Fixed a bug in calculating day, month and year from serial
43   *               number (DG);
44   * 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
45   *               month and year.  Thanks to Trevor Hills for the report (DG);
46   * 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG);
47   * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
48   * 13-Mar-2003 : Implemented Serializable (DG);
49   * 04-Sep-2003 : Completed isInRange() methods (DG);
50   * 05-Sep-2003 : Implemented Comparable (DG);
51   * 21-Oct-2003 : Added hashCode() method (DG);
52   *
53   */
54
55 package org.jfree.date;
56
57 import java.util.Calendar;
58 import java.util.Date;
59
60 /**
61   * Represents a date using an integer, in a similar fashion to the
62   * implementation in Microsoft Excel.  The range of dates supported is
63   * 1-Jan-1900 to 31-Dec-9999.
64   * <P>
65   * Be aware that there is a deliberate bug in Excel that recognises the year
66   * 1900 as a leap year when in fact it is not a leap year. You can find more
67   * information on the Microsoft website in article Q181370:
68   * <P>
69   * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70   * <P>
71   * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
72   * convention 1-Jan-1900 = 2.
73   * The result is that the day number in this class will be different to the
74   * Excel figure for January and February 1900...but then Excel adds in an extra
75   * day (29-Feb-1900 which does not actually exist!) and from that point forward
76   * the day numbers will match.
77   *
78   * @author David Gilbert
79   */
80 public class SpreadsheetDate extends SerialDate {
81
82     /** For serialization. */
83     private static final long serialVersionUID = -2039586705374454461L;
84
85     /**
86       * The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
87       * 2958465).
88       */
89     private int serial;
90
91     /** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
92     private int day;
93
```

```
94      /** The month of the year (1 to 12). */
95      private int month;
96
97      /** The year (1900 to 9999). */
98      private int year;
99
100     /** An optional description for the date. */
101     private String description;
102
103     /**
104      * Creates a new date instance.
105      *
106      * @param day  the day (in the range 1 to 28/29/30/31).
107      * @param month  the month (in the range 1 to 12).
108      * @param year  the year (in the range 1900 to 9999).
109      */
110     public SpreadsheetDate(final int day, final int month, final int year) {
111
112         if ((year >= 1900) && (year <= 9999)) {
113             this.year = year;
114         }
115         else {
116             throw new IllegalArgumentException(
117                 "The 'year' argument must be in range 1900 to 9999."
118             );
119         }
120
121         if ((month >= MonthConstants.JANUARY)
122                 && (month <= MonthConstants.DECEMBER)) {
123             this.month = month;
124         }
125         else {
126             throw new IllegalArgumentException(
127                 "The 'month' argument must be in the range 1 to 12."
128             );
129         }
130
131         if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132             this.day = day;
133         }
134         else {
135             throw new IllegalArgumentException("Invalid 'day' argument.");
136         }
137
138         // the serial number needs to be synchronised with the day-month-year...
139         this.serial = calcSerial(day, month, year);
140
141         this.description = null;
142
143     }
144
145     /**
146      * Standard constructor - creates a new date object representing the
147      * specified day number (which should be in the range 2 to 2958465.
148      *
149      * @param serial  the serial number for the day (range: 2 to 2958465).
150      */
151     public SpreadsheetDate(final int serial) {
152
153         if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154             this.serial = serial;
155         }
156         else {
```

```
157             throw new IllegalArgumentException(
158                 "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159         }
160
161         // the day-month-year needs to be synchronised with the serial number...
162         calcDayMonthYear();
163
164     }
165
166     /**
167      * Returns the description that is attached to the date.  It is not
168      * required that a date have a description, but for some applications it
169      * is useful.
170      *
171      * @return The description that is attached to the date.
172      */
173     public String getDescription() {
174         return this.description;
175     }
176
177     /**
178      * Sets the description for the date.
179      *
180      * @param description  the description for this date (<code>null</code>
181      *                     permitted).
182      */
183     public void setDescription(final String description) {
184         this.description = description;
185     }
186
187     /**
188      * Returns the serial number for the date, where 1 January 1900 = 2
189      * (this corresponds, almost, to the numbering system used in Microsoft
190      * Excel for Windows and Lotus 1-2-3).
191      *
192      * @return The serial number of this date.
193      */
194     public int toSerial() {
195         return this.serial;
196     }
197
198     /**
199      * Returns a <code>java.util.Date</code> equivalent to this date.
200      *
201      * @return The date.
202      */
203     public Date toDate() {
204         final Calendar calendar = Calendar.getInstance();
205         calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206         return calendar.getTime();
207     }
208
209     /**
210      * Returns the year (assume a valid range of 1900 to 9999).
211      *
212      * @return The year.
213      */
214     public int getYYYY() {
215         return this.year;
216     }
217
218     /**
```

```
219       * Returns the month (January = 1, February = 2, March = 3).
220       *
221       * @return The month of the year.
222       */
223      public int getMonth() {
224          return this.month;
225      }
226
227      /**
228       * Returns the day of the month.
229       *
230       * @return The day of the month.
231       */
232      public int getDayOfMonth() {
233          return this.day;
234      }
235
236      /**
237       * Returns a code representing the day of the week.
238       * <P>
239       * The codes are defined in the {@link SerialDate} class as:
240       * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241       * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
242       * <code>SATURDAY</code>.
243       *
244       * @return A code representing the day of the week.
245       */
246      public int getDayOfWeek() {
247          return (this.serial + 6) % 7 + 1;
248      }
249
250      /**
251       * Tests the equality of this date with an arbitrary object.
252       * <P>
253       * This method will return true ONLY if the object is an instance of the
254       * {@link SerialDate} base class, and it represents the same day as this
255       * {@link SpreadsheetDate}.
256       *
257       * @param object  the object to compare (<code>null</code> permitted).
258       *
259       * @return A boolean.
260       */
261      public boolean equals(final Object object) {
262
263          if (object instanceof SerialDate) {
264              final SerialDate s = (SerialDate) object;
265              return (s.toSerial() == this.toSerial());
266          }
267          else {
268              return false;
269          }
270
271      }
272
273      /**
274       * Returns a hash code for this object instance.
275       *
276       * @return A hash code.
277       */
278      public int hashCode() {
279          return toSerial();
280      }
```

```
281
282     /**
283      * Returns the difference (in days) between this date and the specified
284      * 'other' date.
285      *
286      * @param other  the date being compared to.
287      *
288      * @return The difference (in days) between this date and the specified
289      *         'other' date.
290      */
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Implements the method required by the Comparable interface.
297      *
298      * @param other  the other object (usually another SerialDate).
299      *
300      * @return A negative integer, zero, or a positive integer as this object
301      *         is less than, equal to, or greater than the specified object.
302      */
303     public int compareTo(final Object other) {
304         return compare((SerialDate) other);
305     }
306
307     /**
308      * Returns true if this SerialDate represents the same date as the
309      * specified SerialDate.
310      *
311      * @param other  the date being compared to.
312      *
313      * @return <code>true</code> if this SerialDate represents the same date as
314      *         the specified SerialDate.
315      */
316     public boolean isOn(final SerialDate other) {
317         return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Returns true if this SerialDate represents an earlier date compared to
322      * the specified SerialDate.
323      *
324      * @param other  the date being compared to.
325      *
326      * @return <code>true</code> if this SerialDate represents an earlier date
327      *         compared to the specified SerialDate.
328      */
329     public boolean isBefore(final SerialDate other) {
330         return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Returns true if this SerialDate represents the same date as the
335      * specified SerialDate.
336      *
337      * @param other  the date being compared to.
338      *
339      * @return <code>true</code> if this SerialDate represents the same date
340      *         as the specified SerialDate.
341      */
```

```
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Returns true if this SerialDate represents the same date as the
348      * specified SerialDate.
349      *
350      * @param other  the date being compared to.
351      *
352      * @return <code>true</code> if this SerialDate represents the same date
353      *      as the specified SerialDate.
354      */
355     public boolean isAfter(final SerialDate other) {
356         return (this.serial > other.toSerial());
357     }
358
359     /**
360      * Returns true if this SerialDate represents the same date as the
361      * specified SerialDate.
362      *
363      * @param other  the date being compared to.
364      *
365      * @return <code>true</code> if this SerialDate represents the same date as
366      *      the specified SerialDate.
367      */
368     public boolean isOnOrAfter(final SerialDate other) {
369         return (this.serial >= other.toSerial());
370     }
371
372     /**
373      * Returns <code>true</code> if this {@link SerialDate} is within the
374      * specified range (INCLUSIVE).  The date order of d1 and d2 is not
375      * important.
376      *
377      * @param d1  a boundary date for the range.
378      * @param d2  the other boundary date for the range.
379      *
380      * @return A boolean.
381      */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383         return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
386     /**
387      * Returns true if this SerialDate is within the specified range (caller
388      * specifies whether or not the end-points are included).  The order of d1
389      * and d2 is not important.
390      *
391      * @param d1  one boundary date for the range.
392      * @param d2  a second boundary date for the range.
393      * @param include  a code that controls whether or not the start and end
394      *                 dates are included in the range.
395      *
396      * @return <code>true</code> if this SerialDate is within the specified
397      *      range.
398      */
399     public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                              final int include) {
401         final int s1 = d1.toSerial();
402         final int s2 = d2.toSerial();
```

```
403         final int start = Math.min(s1, s2);
404         final int end = Math.max(s1, s2);
405
406         final int s = toSerial();
407         if (include == SerialDate.INCLUDE_BOTH) {
408             return (s >= start && s <= end);
409         }
410         else if (include == SerialDate.INCLUDE_FIRST) {
411             return (s >= start && s < end);
412         }
413         else if (include == SerialDate.INCLUDE_SECOND) {
414             return (s > start && s <= end);
415         }
416         else {
417             return (s > start && s < end);
418         }
419     }
420
421     /**
422      * Calculate the serial number from the day, month and year.
423      * <P>
424      * 1-Jan-1900 = 2.
425      *
426      * @param d  the day.
427      * @param m  the month.
428      * @param y  the year.
429      *
430      * @return the serial number from the day, month and year.
431      */
432     private int calcSerial(final int d, final int m, final int y) {
433         final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434         int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435         if (m > MonthConstants.FEBRUARY) {
436             if (SerialDate.isLeapYear(y)) {
437                 mm = mm + 1;
438             }
439         }
440         final int dd = d;
441         return yy + mm + dd + 1;
442     }
443
444     /**
445      * Calculate the day, month and year from the serial number.
446      */
447     private void calcDayMonthYear() {
448
449         // get the year from the serial date
450         final int days = this.serial - SERIAL_LOWER_BOUND;
451         // overestimated because we ignored leap days
452         final int overestimatedYYYY = 1900 + (days / 365);
453         final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454         final int nonleapdays = days - leaps;
455         // underestimated because we overestimated years
456         int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458         if (underestimatedYYYY == overestimatedYYYY) {
459             this.year = underestimatedYYYY;
460         }
461         else {
462             int ss1 = calcSerial(1, 1, underestimatedYYYY);
463             while (ss1 <= this.serial) {
464                 underestimatedYYYY = underestimatedYYYY + 1;
465                 ss1 = calcSerial(1, 1, underestimatedYYYY);
```

```
466             }
467             this.year = underestimatedYYYY - 1;
468         }
469
470         final int ss2 = calcSerial(1, 1, this.year);
471
472         int[] daysToEndOfPrecedingMonth
473             = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475         if (isLeapYear(this.year)) {
476             daysToEndOfPrecedingMonth
477                 = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478         }
479
480         // get the month from the serial date
481         int mm = 1;
482         int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483         while (sss < this.serial) {
484             mm = mm + 1;
485             sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486         }
487         this.month = mm - 1;
488
489         // what's left is d(+1);
490         this.day = this.serial - ss2
491                    - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

LISTING B6. RelativeDayOfWeekRule.java

```
1  /* ===========================================================
2   * JCommon : a free general purpose class library for the Java(tm) platform
3   * ===========================================================
4   *
5   * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6   *
7   * Project Info:  http://www.jfree.org/jcommon/index.html
8   *
9   * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * --------------------------
28  * RelativeDayOfWeekRule.java
29  * --------------------------
```

```
 * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: RelativeDayOfWeekRule.java,v 1.5 2005/11/03 09:24:45 mungady Exp $
 *
 * Changes (from 26-Oct-2001)
 * --------------------------
 * 26-Oct-2001 : Changed package to com.jrefinery.date.*;
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 *
 */

package org.jfree.date;

/**
 * An annual date rule that returns a date for each year based on (a) a
 * reference rule; (b) a day of the week; and (c) a selection parameter
 * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
 * <P>
 * For example, Good Friday can be specified as 'the Friday PRECEDING Easter
 * Sunday'.
 *
 * @author David Gilbert
 */
public class RelativeDayOfWeekRule extends AnnualDateRule {

    /** A reference to the annual date rule on which this rule is based. */
    private AnnualDateRule subrule;

    /**
     * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).
     */
    private int dayOfWeek;

    /** Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). */
    private int relative;

    /**
     * Default constructor - builds a rule for the Monday following 1 January.
     */
    public RelativeDayOfWeekRule() {
        this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
    }

    /**
     * Standard constructor - builds rule based on the supplied sub-rule.
     *
     * @param subrule  the rule that determines the reference date.
     * @param dayOfWeek  the day-of-the-week relative to the reference date.
     * @param relative  indicates *which* day-of-the-week (preceding, nearest
     *                  or following).
     */
    public RelativeDayOfWeekRule(final AnnualDateRule subrule,
            final int dayOfWeek, final int relative) {
        this.subrule = subrule;
        this.dayOfWeek = dayOfWeek;
        this.relative = relative;
    }

```

```
91      /**
92       * Returns the sub-rule (also called the reference rule).
93       *
94       * @return The annual date rule that determines the reference date for this
95       *      rule.
96       */
97      public AnnualDateRule getSubrule() {
98          return this.subrule;
99      }
100
101     /**
102      * Sets the sub-rule.
103      *
104      * @param subrule  the annual date rule that determines the reference date
105      *          for this rule.
106      */
107     public void setSubrule(final AnnualDateRule subrule) {
108         this.subrule = subrule;
109     }
110
111     /**
112      * Returns the day-of-the-week for this rule.
113      *
114      * @return the day-of-the-week for this rule.
115      */
116     public int getDayOfWeek() {
117         return this.dayOfWeek;
118     }
119
120     /**
121      * Sets the day-of-the-week for this rule.
122      *
123      * @param dayOfWeek  the day-of-the-week (SerialDate.MONDAY,
124      *           SerialDate.TUESDAY, and so on).
125      */
126     public void setDayOfWeek(final int dayOfWeek) {
127         this.dayOfWeek = dayOfWeek;
128     }
129
130     /**
131      * Returns the 'relative' attribute, that determines *which*
132      * day-of-the-week we are interested in (SerialDate.PRECEDING,
133      * SerialDate.NEAREST or SerialDate.FOLLOWING).
134      *
135      * @return The 'relative' attribute.
136      */
137     public int getRelative() {
138         return this.relative;
139     }
140
141     /**
142      * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
143      * SerialDate.FOLLOWING).
144      *
145      * @param relative  determines *which* day-of-the-week is selected by this
146      *          rule.
147      */
148     public void setRelative(final int relative) {
149         this.relative = relative;
150     }
151
```

```
152     /**
153      * Creates a clone of this rule.
154      *
155      * @return a clone of this rule.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167      * Returns the date generated by this rule, for the specified year.
168      *
169      * @param year  the year (1900 &lt;= year &lt;= 9999).
170      *
171      * @return The date generated by the rule for the given year (possibly
172      *     <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // check argument...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // calculate the date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                     result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                             base);
192                     break;
193                 case(SerialDate.NEAREST):
194                     result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                             base);
196                     break;
197                 case(SerialDate.FOLLOWING):
198                     result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                             base);
200                     break;
201                 default:
202                     break;
203             }
204         }
205         return result;
206
207     }
208
209 }
```

LISTING B7. DayDate.java (ostateczny)

```
 1 /* ===========================================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
...
36  */
37 package org.jfree.date;
38
39 import java.io.Serializable;
40 import java.util.*;
41
42 /**
43  * Klasa abstrakcyjna, która reprezentuje niezmienne daty z dokładnością
44  * do jednego dnia. Implementacja odwzorowuje każdy dzień na liczbę integer
45  * reprezentującą kolejny dzień od pewnego dnia początkowego.
46  *
47  * Dlaczego nie skorzystać po prostu z java.util.Date? Robimy to, gdy ma to sens.
48  * Czasami java.util.Date może być *zbyt* dokładna - reprezentuje czas
49  * z dokładnością do 1/1000 sekundy (a sama data jest zależna od strefy czasowej).
50  * Czasami chcemy reprezentować określony dzień (np. 21 stycznia 2015)
51  * bez zajmowania się czasem w tym dniu, strefą czasową
52  * czy też innymi parametrami. Do tego właśnie wykorzystujemy DayDate.
53  *
54  * Do utworzenia obiektu będziemy używać DayDateFactory.makeDate.
55  *
56  * @author David Gilbert
57  * @author Robert C. Martin znacznie przebudował kod.
58  */
59
60 public abstract class DayDate implements Comparable, Serializable {
61   public abstract int getOrdinalDay();
62   public abstract int getYear();
63   public abstract Month getMonth();
64   public abstract int getDayOfMonth();
65
66   protected abstract Day getDayOfWeekForOrdinalZero();
67
68   public DayDate plusDays(int days) {
69     return DayDateFactory.makeDate(getOrdinalDay() + days);
70   }
71
72   public DayDate plusMonths(int months) {
73     int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74     int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75     int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
76     int resultYear = resultMonthAndYearAsOrdinal / 12;
77     int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
78     Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79     int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
80     return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81   }
82
83   public DayDate plusYears(int years) {
84     int resultYear = getYear() + years;
85     int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
86     return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87   }
88
89   private int correctLastDayOfMonth(int day, Month month, int year) {
```

```
90      int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
91      if (day > lastDayOfMonth)
92        day = lastDayOfMonth;
93      return day;
94    }
95
96    public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
97      int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
98      if (offsetToTarget >= 0)
99        offsetToTarget -= 7;
100     return plusDays(offsetToTarget);
101   }
102
103   public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105     if (offsetToTarget <= 0)
106       offsetToTarget += 7;
107     return plusDays(offsetToTarget);
108   }
109
110   public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111     int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112     int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113     int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115     if (offsetToFutureTarget > 3)
116       return plusDays(offsetToPreviousTarget);
117     else
118       return plusDays(offsetToFutureTarget);
119   }
120
121   public DayDate getEndOfMonth() {
122     Month month = getMonth();
123     int year = getYear();
124     int lastDay = DateUtil.lastDayOfMonth(month, year);
125     return DayDateFactory.makeDate(lastDay, month, year);
126   }
127
128   public Date toDate() {
129     final Calendar calendar = Calendar.getInstance();
130     int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131     calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132     return calendar.getTime();
133   }
134
135   public String toString() {
136     return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137   }
138
139   public Day getDayOfWeek() {
140     Day startingDay = getDayOfWeekForOrdinalZero();
141     int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142     int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143     return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144   }
145
146   public int daysSince(DayDate date) {
147     return getOrdinalDay() - date.getOrdinalDay();
148   }
149
150   public boolean isOn(DayDate other) {
151     return getOrdinalDay() == other.getOrdinalDay();
152   }
153
```

```
154   public boolean isBefore(DayDate other) {
155     return getOrdinalDay() < other.getOrdinalDay();
156   }
157
158   public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
160   }
161
162   public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164   }
165
166   public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168   }
169
170   public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172   }
173
174   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178   }
179 }
```

LISTING B8. Month.java (ostateczny)

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public enum Month {
 6   JANUARY(1), FEBRUARY(2), MARCH(3),
 7   APRIL(4),   MAY(5),      JUNE(6),
 8   JULY(7),    AUGUST(8),   SEPTEMBER(9),
 9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11   private static final int[] LAST_DAY_OF_MONTH =
12   {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14   private int index;
15
16   Month(int index) {
17     this.index = index;
18   }
19
20   public static Month fromInt(int monthIndex) {
21     for (Month m : Month.values()) {
22       if (m.index == monthIndex)
23         return m;
24     }
25     throw new IllegalArgumentException("Niewłaściwy indeks miesiąca " + monthIndex);
26   }
27
28   public int lastDay() {
29     return LAST_DAY_OF_MONTH[index];
30   }
31
32   public int quarter() {
33     return 1 + (index - 1) / 3;
34   }
```

```
35
36   public String toString() {
37     return dateFormatSymbols.getMonths()[index - 1];
38   }
39
40   public String toShortString() {
41     return dateFormatSymbols.getShortMonths()[index - 1];
42   }
43
44   public static Month parse(String s) {
45     s = s.trim();
46     for (Month m : Month.values())
47       if (m.matches(s))
48         return m;
49
50     try {
51       return fromInt(Integer.parseInt(s));
52     }
53     catch (NumberFormatException e) {}
54       throw new IllegalArgumentException("Nieprawidłowy miesiąc " + s);
55   }
56
57   private boolean matches(String s) {
58     return s.equalsIgnoreCase(toString()) ||
59     s.equalsIgnoreCase(toShortString());
60   }
61
62   public int toInt() {
63     return index;
64   }
65 }
```

LISTING B9. Day.java (ostateczny)

```
 1 package org.jfree.date;
 2
 3 import java.util.Calendar;
 4 import java.text.DateFormatSymbols;
 5
 6 public enum Day {
 7   MONDAY(Calendar.MONDAY),
 8   TUESDAY(Calendar.TUESDAY),
 9   WEDNESDAY(Calendar.WEDNESDAY),
10   THURSDAY(Calendar.THURSDAY),
11   FRIDAY(Calendar.FRIDAY),
12   SATURDAY(Calendar.SATURDAY),
13   SUNDAY(Calendar.SUNDAY);
14
15   private final int index;
16   private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18   Day(int day) {
19     index = day;
20   }
21
22   public static Day fromInt(int index) throws IllegalArgumentException {
23     for (Day d : Day.values())
24       if (d.index == index)
25         return d;
26     throw new IllegalArgumentException(
27       String.format("Nieprawidłowy indeks dnia: %d.", index));
28   }
29
```

```
30   public static Day parse(String s) throws IllegalArgumentException {
31     String[] shortWeekdayNames =
32     dateSymbols.getShortWeekdays();
33     String[] weekDayNames =
34     dateSymbols.getWeekdays();
35
36     s = s.trim();
37     for (Day day : Day.values()) {
38       if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39           s.equalsIgnoreCase(weekDayNames[day.index])) {
40         return day;
41       }
42     }
43     throw new IllegalArgumentException(
44     String.format("%s nie jest prawidłową nazwą dnia tygodnia", s));
45   }
46
47   public String toString() {
48     return dateSymbols.getWeekdays()[index];
49   }
50
51   public int toInt() {
52     return index;
53   }
54 }
```

LISTING B10. DateInterval.java (ostateczny)

```
 1 package org.jfree.date;
 2
 3 public enum DateInterval {
 4   OPEN {
 5     public boolean isIn(int d, int left, int right) {
 6       return d > left && d < right;
 7     }
 8   },
 9   CLOSED_LEFT {
10     public boolean isIn(int d, int left, int right) {
11       return d >= left && d < right;
12     }
13   },
14   CLOSED_RIGHT {
15     public boolean isIn(int d, int left, int right) {
16       return d > left && d <= right;
17     }
18   },
19   CLOSED {
20     public boolean isIn(int d, int left, int right) {
21       return d >= left && d <= right;
22     }
23   };
24
25   public abstract boolean isIn(int d, int left, int right);
26 }
```

LISTING B11. WeekInMonth.java (ostateczny)

```
 1 package org.jfree.date;
 2
 3 public enum WeekInMonth {
 4   FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
 5   private final int index;
 6
```

```
 7   WeekInMonth(int index) {
 8     this.index = index;
 9   }
10
11   public int toInt() {
12     return index;
13   }
14 }
```

LISTING B12. WeekdayRange.java (ostateczny)

```
1 package org.jfree.date;
2
3 public enum WeekdayRange {
4   LAST, NEAREST, NEXT
5 }
```

LISTING B13. DateUtil.java (ostateczny)

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public class DateUtil {
 6   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
 7
 8   public static String[] getMonthNames() {
 9     return dateFormatSymbols.getMonths();
10   }
11
12   public static boolean isLeapYear(int year) {
13     boolean fourth = year % 4 == 0;
14     boolean hundredth = year % 100 == 0;
15     boolean fourHundredth = year % 400 == 0;
16     return fourth && (!hundredth || fourHundredth);
17   }
18
19   public static int lastDayOfMonth(Month month, int year) {
20     if (month == Month.FEBRUARY && isLeapYear(year))
21       return month.lastDay() + 1;
22     else
23       return month.lastDay();
24   }
25
26   public static int leapYearCount(int year) {
27     int leap4 = (year - 1896) / 4;
28     int leap100 = (year - 1800) / 100;
29     int leap400 = (year - 1600) / 400;
30     return leap4 - leap100 + leap400;
31   }
32 }
```

LISTING B14. DayDateFactory.java (ostateczny)

```
 1 package org.jfree.date;
 2
 3 public abstract class DayDateFactory {
 4   private static DayDateFactory factory = new SpreadsheetDateFactory();
 5   public static void setInstance(DayDateFactory factory) {
 6     DayDateFactory.factory = factory;
 7   }
 8
 9   protected abstract DayDate _makeDate(int ordinal);
```

```
  protected abstract DayDate _makeDate(int day, Month month, int year);
  protected abstract DayDate _makeDate(int day, int month, int year);
  protected abstract DayDate _makeDate(java.util.Date date);
  protected abstract int _getMinimumYear();
  protected abstract int _getMaximumYear();

  public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
  }

  public static DayDate makeDate(int day, Month month, int year) {
    return factory._makeDate(day, month, year);
  }

  public static DayDate makeDate(int day, int month, int year) {
    return factory._makeDate(day, month, year);
  }

  public static DayDate makeDate(java.util.Date date) {
    return factory._makeDate(date);
  }

  public static int getMinimumYear() {
    return factory._getMinimumYear();
  }

  public static int getMaximumYear() {
    return factory._getMaximumYear();
  }
}
```

LISTING B15. SpreadsheetDateFactory.java (ostateczny)

```
package org.jfree.date;

import java.util.*;

public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }

  public DayDate _makeDate(int day, Month month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
      calendar.get(Calendar.DATE),
      Month.fromInt(calendar.get(Calendar.MONTH) + 1),
      calendar.get(Calendar.YEAR));
  }

  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }

```

```
31   protected int _getMaximumYear() {
32     return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33   }
34 }
```

LISTING B16. SpreadsheetDate.java (ostateczny)

```
  1 /*
==========================================================================
  2 * JCommon : a free general purpose class library for the Java(tm) platform
  3 *
==========================================================================
  4 *
  5 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6 *
...
 52 *
 53 */
 54
 55 package org.jfree.date;
 56
 57 import static org.jfree.date.Month.FEBRUARY;
 58
 59 import java.util.*;
 60
 61 /**
 62  * Reprezentuje daty przy użyciu liczb integer, podobnie jak w
 63  * programie Microsoft Excel. Obsługiwany zakres dat: od
 64  * 1 stycznia 1900 do 31 grudnia 9999.
 65  * <p/>
 66  * Trzeba zwrócić uwagę, że w Excelu występuje błąd, powodujący uznanie roku
 67  * 1900 za rok przestępny, choć nim nie jest. Więcej informacji na ten temat
 68  * można znaleźć w witrynie Microsoft, w artykule Q181370:
 69  * <p/>
 70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 71  * <p/>
 72  * Excel korzysta z konwencji, w  której 1 stycznia 1900 = 1. Ta klasa korzysta
 73  * z konwencji, w której 1 stycznia 1900 = 2.
 74  * W wyniku tego numer dnia w tej klasie jest inny niż w
 75  * Excelu dla stycznia i lutego 1900..., ale wtedy Excel wprowadza dodatkowy dzień
 76  * (29 lutego 1900, który faktycznie nie istniał!) i od tego momentu
 77  * numery dni zgadzają się.
 78  *
 79  * @author David Gilbert
 80 */
 81 public class SpreadsheetDate extends DayDate {
 82   public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
 83   public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
 84   public static final int MINIMUM_YEAR_SUPPORTED = 1900;
 85   public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
 86   static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
 88   static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
 90
 91   private int ordinalDay;
 92   private int day;
 93   private Month month;
 94   private int year;
 95
 96   public SpreadsheetDate(int day, Month month, int year) {
 97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
```

```
 98       throw new IllegalArgumentException(
 99         "Argument 'year' musi zawierać się w zakresie " +
100          MINIMUM_YEAR_SUPPORTED + " do " + MAXIMUM_YEAR_SUPPORTED + ".");
101      if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102        throw new IllegalArgumentException("Nieprawidłowy argument 'day'.");
103
104      this.year = year;
105      this.month = month;
106      this.day = day;
107      ordinalDay = calcOrdinal(day, month, year);
108      }
109
110    public SpreadsheetDate(int day, int month, int year) {
111      this(day, Month.fromInt(month), year);
112    }
113
114    public SpreadsheetDate(int serial) {
115      if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116        throw new IllegalArgumentException(
117          "SpreadsheetDate: wartość 'serial' musi być z zakresu od 2 do 2958465.");
118
119      ordinalDay = serial;
120      calcDayMonthYear();
121    }
122
123    public int getOrdinalDay() {
124      return ordinalDay;
125    }
126
127    public int getYear() {
128      return year;
129    }
130
131    public Month getMonth() {
132      return month;
133    }
134
135    public int getDayOfMonth() {
136      return day;
137    }
138
139    protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141    public boolean equals(Object object) {
142      if (!(object instanceof DayDate))
143        return false;
144
145      DayDate date = (DayDate) object;
146      return date.getOrdinalDay() == getOrdinalDay();
147    }
148
149    public int hashCode() {
150      return getOrdinalDay();
151    }
152
153    public int compareTo(Object other) {
154      return daysSince((DayDate) other);
155    }
156
157    private int calcOrdinal(int day, Month month, int year) {
158      int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159      int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160      int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161      if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
```

```
162         daysUpToMonth++;
163       int daysInMonth = day - 1;
164       return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165     }
166
167     private void calcDayMonthYear() {
168       int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169       int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170       int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171       int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173       year = huntForYearContaining(ordinalDay, underestimatedYear);
174       int firstOrdinalOfYear = firstOrdinalOfYear(year);
175       month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176       day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177     }
178
179     private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180       int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181       int aMonth = 1;
182       while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183         aMonth++;
184
185       return Month.fromInt(aMonth - 1);
186     }
187
188     private int daysBeforeThisMonth(int aMonth) {
189       if (DateUtil.isLeapYear(year))
190         return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191       else
192         return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193     }
194
195     private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196       int aYear = startingYear;
197       while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198         aYear++;
199
200       return aYear - 1;
201     }
202
203     private int firstOrdinalOfYear(int year) {
204       return calcOrdinal(1, Month.JANUARY, year);
205     }
206
207     public static DayDate createInstance(Date date) {
208       GregorianCalendar calendar = new GregorianCalendar();
209       calendar.setTime(date);
210       return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                  Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                  calendar.get(Calendar.YEAR));
213
214     }
215 }
```

DODATEK C

Odwołania do heurystyk

ODWOŁANIA DO *ZAPACHÓW* KODU I HEURYSTYK. Pozostałe odwołania mogą być usunięte.

Epilog

W ROKU 2005, W CZASIE konferencji Agile w Denver, Elisabeth Hedrickson[1] podała mi zieloną opaskę na rękę, podobną do tej, którą tak spopularyzował Lance Armstrong. Moja miała napis „Test Obsessed" (ogarnięty obsesją testów). Chętnie ją założyłem i nosiłem z dumą. Od momentu nauczenia się TDD od Kenta Becka, w roku 1999, faktycznie stałem się entuzjastą programowania sterowanego testami.

Jednak wtedy stało się coś dziwnego. Odkryłem, że nie mogę zdjąć tej opaski. Nie dlatego, że fizycznie się zablokowała. Opaska ta ujawniała moją profesjonalną etykę. Była widocznym wskaźnikiem mojego zaangażowania w tworzenie najlepszego kodu, jaki mogłem napisać. Jej zdjęcie wydawało się zdradą tej etyki, a na to nie mogłem sobie pozwolić.

Dlatego nadal jest ona na moim nadgarstku. Gdy piszę kod, widzę ją kątem oka. Jest to stałe przypomnienie celu, jaki sobie postawiłem — celem tym jest pisanie czystego kodu.

[1] *http://www.qualitytree.com*

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

Ź

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA