

# čistý kód

NÁVRHOVÉ VZORY,  
REFAKTOROVÁNÍ, TESTOVÁNÍ  
A DALŠÍ TECHNIKY AGILNÍHO  
PROGRAMOVÁNÍ

Určeno pro vývojáře, softwarové inženýry, projektové manažery, vedoucí týmů a systémové analytiky

Poznáte rozdíl mezi špatným a správným kódem?

Používáte správné názvy, třídy, funkce nebo objekty?

Umíte řešit problémy bez obcházení logiky kódu?



# čistý kód

NÁVRHOVÉ VZORY,  
REFAKTOROVÁNÍ, TESTOVÁNÍ  
A DALŠÍ TECHNIKY AGILNÍHO  
PROGRAMOVÁNÍ

Poznáte špatný kód od dobrého? Naučte se tvořit správný a srozumitelný kód nejen pro efektivní týmovou spolupráci. Zjistěte, jak opravit špatný kód na správný. Osvojíte si tak návyky a postupy profesionálů v oboru.

Kniha se v jednotlivých kapitolách zaměřuje na časté problémy, se kterými se lze setkat při psaní kódu v libovolném jazyce. Prozradí vám, čemu se vyhnout, které vlastnosti by měl kód mít, a také nabídne celou řadu **profesionálních doporučení**, jak průběžně zlepšovat opakovaně požívaný kód. Obecné rady, které lze aplikovat na libovolný jazyk, doplňují ukázky v Javě.

Publikace vás mimo jiné naučí, jak:

- Vybírat srozumitelné názvy funkcí, tříd, metod a objektů
- Správně rozložit funkčnost projektu mezi funkce
- Vytvářet hodnotné komentáře
- Formátovat kód pro co nejlepší čitelnost
- Efektivně zpracovávat chyby
- Testovat právě vytvářený projekt
- Využít paralelního zpracování k lepšímu využití hardwaru
- Zlepšit nebo opravit již vytvořený kód

Publikace je určena **programátorem**, **softwarovým inženýrem**, **vedoucím týmu**, **projektovým manažerem** nebo **systémovým analytikem**.

## O autorovi:

**Robert C. Martin** je profesionální softwarový poradce, zakladatel a prezident Object Mentor, Inc. Tuto společnost tvoří tým zkušených poradců a školitelů v programovacích jazycích C++, Java, C#, Ruby a technikách agilního programování, návrhových vzorů a extrémního programování.

## Zařazení publikace:

	začátečník	pokročilý	odborník
student			
IT manažer			
programátor			
analytik			

**Computer Press, a.s.**

Holandská 8  
639 00 Brno

Objednávejte na:  
<http://knihy.cpress.cz>  
distribuce@cpress.cz

Bezplatná linka  
**800 555 513**

**C P R E S S**



ISBN 978-80-251-2285-3

Prodejní kód: K1646



**C P R E S S**

v kompletní nabídce najeznete  
již více než 2200 knižních titulů  
z mnoha zajímavých oborů

Navštívte  
náš  
web!

9 788025 122853

# Čistý kód

**Robert C. Martin**

**Computer Press, a.s.**, 2009. Vydání první.

**Překlad:** Jiří Berka

**Odborná korektura:** Miroslav Virius

**Jazyková korektura:** Pavel Bubla

**Sazba:** Martina Petrová

**Rejstřík:** Daniel Štreit

**Obálka:** Martin Sodomka

**Komentář na zadní straně obálky:** Martin Herodek

**Technická spolupráce:** Jiří Matoušek,

Dagmar Hajdajová

**Odpovědný redaktor:** Martin Herodek

**Technický redaktor:** Jiří Matoušek

**Produkce:** Petr Baláš

Authorized translation from the English language edition, entitled CLEAN CODE: A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP, 1st Edition, 0132350882 by MARTIN, ROBERT C., published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2009 Pearson Education, Inc.

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. CZECH language edition published by COMPUTER PRESS, A.S., Copyright © 2009.

Autorizovaný překlad z originálního anglického vydání Clean Code: A Handbook of Agile Software Craftsmanship.

Originální copyright: published by Pearson Education, Inc, publishing as Prentice Hall, Copyright ©2009.

Překlad: © Computer Press, a.s., 2009.

**Computer Press, a.s.,**  
Holandská 8, 639 00 Brno

Objednávky knih:

<http://knihy.cpress.cz>

distribuce@cpress.cz

tel.: 800 555 513

ISBN 978-80-251-2285-3

Prodejný kód: K1646

Vydalo nakladatelství Computer Press, a.s., jako svou 3318. publikaci.

© Computer Press, a.s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopírována a rozmnožována za účelem rozšiřování v jakémkoli formě či jakýmkoli způsobem bez písemného souhlasu vydavatele.

# Stručný obsah

Předmluva .....	17
Úvod .....	21
Čistý kód .....	25
Smysluplná jména .....	39
Funkce .....	53
Komentáře .....	73
Formátování .....	95
Objekty a datové struktury .....	111
Zpracování chyb .....	121
Hranice .....	131
Jednotkové testy .....	139
Třídy .....	151
Systémy .....	167
Vývoj .....	183
Souběžnost .....	189
Postupné vylepšování .....	203
Vnitřní části šablony JUnit .....	259
Refaktorování třídy <code>Serializable</code> .....	273
Skryté problémy a heuristika .....	291
Souběžnost II .....	321
Doslov .....	415

# Obsah

Předmluva .....	17
Úvod .....	21
Poděkování.....	23
Poznámka redakce českého vydání .....	23

## Kapitola 1

### **Čistý kód.....25**

Kód nezanikne .....	26
Špatný kód .....	26
Celková cena za nepořádek .....	27
Celková rekonstrukce na zelené louce .....	28
Přístup .....	28
Prvotní paradox .....	29
Umění čistého kódu?.....	29
Co je to čistý kód?.....	30
Myšlenkový směr .....	34
Autoři.....	35
Skautské pravidlo .....	36
Úvod a principy .....	37
Závěr.....	37
Použitá literatura.....	37

## Kapitola 2

### **Smysluplná jména .....39**

Úvod .....	40
Používejte jména vysvětlující význam .....	40
Vyhnete se dezinformacím .....	41
Dělejte smysluplné rozdíly .....	42
Používejte vyslovitelná jména .....	43
Používejte jména, která lze vyhledat.....	44
Vyhnete se kódování jmen.....	45
Maďarská notace .....	45
Členské předpony .....	45

---

Rozhraní a implementace.....	46
Vyhnezte se skrytému překládání jmen.....	46
Jména tříd.....	47
Jména metod .....	47
Nesnažte se být strojení .....	47
Volte jedno slovo pro jeden pojem .....	47
Nepoužívejte slovní hříčky .....	48
Používejte jména z domény řešení .....	48
Používejte jména domén problému.....	49
Přidejte smysluplné souvislosti .....	49
Nepřidávejte kontext bezdůvodně .....	50
Slovo na závěr .....	51

## Kapitola 3

### Funkce ..... 53

Malá! .....	56
Bloky a odsazování.....	57
Dělejte jen jednu věc.....	57
Sekce uvnitř funkcí.....	58
Jedna úroveň abstrakce na funkci.....	58
Čtení kódu odshora dolů: metoda sestupu .....	58
Příkazy Switch .....	59
Používejte popisná jména .....	61
Argumenty funkcí.....	61
Běžné tvary funkce s jedním argumentem.....	62
Logické argumenty.....	62
Funkce se dvěma argumenty.....	63
Funkce se třemi argumenty.....	63
Objekty jako argumenty .....	64
Seznamy argumentů.....	64
Slovesa a klíčová slova.....	64
Žádné vedlejší efekty .....	65
Výstupní argumenty.....	65
Oddělování příkazů a dotazů .....	66
Dejte přednost výjimkám před vracením chybových kódů .....	66
Extrahuje bloky Try/Catch.....	67
Zpracování chyb je jedna věc .....	68
Magnet závislosti Error.java .....	68
Neopakujte se .....	68

Strukturované programování.....	69
Jak napíšete funkci, jako je tato? .....	69
Závěr.....	70
Použitá literatura.....	72

## Kapitola 4

### Komentáře..... **73**

Komentáře nevyváží špatný kód.....	75
Vyjádřete se kódem .....	75
Dobré komentáře .....	75
Komentáře právnického charakteru .....	75
Informativní komentáře .....	76
Vysvětlení záměru .....	76
Objasnění.....	77
Varování před důsledky.....	78
Komentáře TODO (co dělat).....	78
Zvýraznění.....	79
Javadoc ve veřejných API.....	79
Špatné komentáře .....	79
Huhňání.....	79
Nadbytečné komentáře .....	80
Matoucí komentáře .....	82
Závazné komentáře .....	82
Deníkové komentáře .....	82
Komentáře obsahující šum .....	83
Rušení nahánějící hrůzu .....	85
Nepoužívejte komentář, když můžete použít funkci nebo proměnnou .....	86
Označení pozice .....	86
Komentáře na konci složených závorek.....	86
Připisování a podtitulky se jmény .....	87
Zakomentované řádky kódu .....	87
Komentáře ve formátu HTML.....	88
Nelokální informace .....	89
Příliš mnoho informací.....	89
Nejasná spojitost .....	89
Záhlaví funkcí .....	90
Javadoc v neveřejném kódu.....	90
Příklad.....	90
Použitá literatura.....	93

**Kapitola 5****Formátování.....95**

Důvody formátování.....	96
Vertikální formátování .....	96
Přirovnání k novinám .....	97
Vertikální oddělování pojmu.....	98
Vertikální hustota .....	99
Vertikální vzdálenost.....	99
Vertikální uspořádání .....	103
Horizontální formátování.....	104
Horizontální oddělování a hustota .....	104
Horizontální zarovnání .....	105
Odsazování.....	106
Prázdné obory.....	108
Týmová pravidla .....	108
Formátovací pravidla strýčka Boba.....	109

**Kapitola 6****Objekty a datové struktury .....111**

Datové abstrakce .....	112
Datová a objektová antisimetrie .....	113
Démétřin zákon .....	115
Vykolejený vlak.....	116
Hybridy.....	116
Skrytá struktura .....	117
Objekty pro přenos dat .....	117
Aktivní záznam .....	118
Shrnutí .....	119
Použitá literatura.....	119

**Kapitola 7****Zpracování chyb .....121**

Používejte výjimky raději než návratové kódy.....	122
Pište nejdříve příkazy Try-Catch-Finally.....	123
Používejte nekontrolované výjimky .....	124
Poskytujte kontext s výjimkami.....	125
Definujte třídy výjimek z hlediska potřeb volajícího .....	125
Definujte normální tok .....	127

Nevracejte hodnotu null .....	128
Nepředávejte hodnotu null .....	129
Závěr .....	130
Použitá literatura .....	130

## Kapitola 8

### **Hranice .....** **131**

Použití kódu třetí strany .....	132
Zkoumání a studium hranic .....	134
Studium log4j .....	134
Poznávací testy se vyplatí .....	136
Používání kódu, který zatím neexistuje .....	136
Čisté hranice .....	137
Použitá literatura .....	138

## Kapitola 9

### **Jednotkové testy .....** **139**

Tři zákony vývoje řízeného testy (TDD) .....	140
Mějte testy čisté .....	141
Testy otevírají další možnosti .....	142
Čisté testy .....	142
Doménově specifický testovací jazyk .....	145
Dvojí standard .....	145
Jedna aserce na jeden test .....	147
Jedna myšlenka pro jeden test .....	148
F.I.R.S.T. .....	149
Závěr .....	149
Použitá literatura .....	150

## Kapitola 10

### **Třídy .....** **151**

Organizace třídy .....	152
Zapouzdření .....	152
Třídy by měly být malé! .....	152
Princip jediné odpovědnosti .....	154
Soudržnost .....	156
Soudržnost vede k mnoho malým třídám .....	156
Organizace podporující změny .....	162

---

Izolování od změn.....	164
Použitá literatura.....	165

## Kapitola 11

### Systémy ..... 167

Jak byste postavili město? .....	168
Oddělujte tvorbu systému od jeho používání .....	168
Separování modulu Main .....	169
Továrny .....	170
Vkládání závislostí .....	170
Škálování.....	171
Průnik zájmů.....	173
Zprostředkovatel v Javě .....	174
Čisté rámce Java AOP .....	176
Aspekty AspectJ .....	179
Testování systémové architektury .....	179
Optimalizujte rozhodování .....	180
Používejte rozumně standardy, pokud přináší prokazatelnou hodnotu	180
Systémy potřebují doménově specifické jazyky .....	181
Závěr.....	181
Použitá literatura.....	182

## Kapitola 12

### Vývoj..... 183

Čistota pomocí vyvíjejícího se návrhu.....	184
První pravidlo jednoduchého návrhu: Projde všemi testy.....	184
Zásady jednoduchého návrhu 2–4: refaktorování .....	185
Žádný zdvojený kód .....	185
Expresivita .....	187
Minimální třídy a metody .....	188
Závěr.....	188
Použitá literatura.....	188

## Kapitola 13

### Souběžnost ..... 189

Proč souběžnost? .....	190
Mýty a mylné názory .....	191
Problémy .....	192

---

Principy ochrany souběžnosti.....	192
Princip jedné odpovědnosti .....	192
Důsledek: omezujte rozsah dat.....	193
Důsledek: používejte kopie dat.....	193
Důsledek: Podprocesy by měly být co nejméně závislé .....	193
Vyznejte se ve své knihovně.....	194
Kolekce, které jsou z hlediska souběžného kódu bezpečné .....	194
Poznejte své běhové modely .....	194
Producent-spotřebitel .....	195
Čtenáři-zapisovatelé .....	195
Stolující filozofové .....	196
Pozor na závislosti mezi synchronizovanými metodami .....	196
Mějte synchronizované sekce malé .....	196
Je obtížné napsat korektní kód pro vypínání.....	197
Testování kódu podprocesů .....	197
Berte nejasná selhání jako budoucí možné problémy podprocesů .....	198
Uvedte nejdříve do provozu kód bez podprocesů .....	198
Vytvářejte souběžný kód jako zásuvný modul .....	198
Vytvořte souběžný kód nastavitelný .....	198
Spouštějte více procesů, než máte procesorů .....	199
Spouštějte kód na různých platformách.....	199
Upravte kód tak, aby vyzkoušel a navodil selhání.....	199
Ruční kódování.....	200
Automatizované kódování.....	200
Závěr.....	201
Použitá literatura.....	202

## Kapitola 14

<b>Postupné vylepšování.....</b>	<b>203</b>
Rozbor analyzátoru argumentů příkazového řádku.....	204
Implementace třídy Args .....	204
Jak jsem to udělal? .....	210
Třída Args: nanečisto .....	210
Tak jsem se zastavil .....	220
O postupných změnách .....	220
Řetězcové argumenty .....	222
Závěr .....	257

## Kapitola 15

### Vnitřní části šablony JUnit ..... 259

Šablona JUnit .....	260
Závěr .....	272

## Kapitola 16

### Refaktorování třídy **SerialDate** ..... 273

Nejdřív ať to funguje .....	274
Pak to sprav.....	276
Závěr.....	289
Použitá literatura.....	289

## Kapitola 17

### Skryté problémy a heuristika ..... 291

Komentáře.....	292
K1: Nevhodné komentáře.....	292
K2: Zastaralé komentáře .....	292
K3: Nadbytečný komentář.....	292
K4: Špatně napsaný komentář .....	293
K5: Zakomentovaný kód.....	293
Prostředí .....	293
P1: Sestavení vyžaduje více než jeden krok.....	293
P2: Testy vyžadují více než jeden krok.....	293
Funkce .....	293
F1: Příliš mnoho argumentů .....	293
F2: Výstupní argumenty .....	294
F3: Logické argumenty.....	294
F4: Mrtvé funkce .....	294
Obecné .....	294
O1: Více jazyků v jednom zdrojovém souboru .....	294
O2: Není implementováno to, co je samozřejmě .....	294
O3: Nekorektní funkčnost na hranicích .....	295
O4: Zrušená zabezpečení .....	295
O5: Zdvojení .....	295
O6: Kód na špatné úrovni abstrakce .....	296
O7: Základní třídy, které závisí na odvozených třídách .....	297
O8: Příliš mnoho informací.....	297
O9: Mrtvý kód .....	297

O10: Vertikální oddělování.....	298
O11: Nekonzistentnost .....	298
O12: Zaneráděnost.....	298
O13: Umělé vazby.....	298
O14: Chybějící schopnosti.....	298
O15: Přepínací argumenty .....	299
O16: Nejasný záměr.....	300
O17: Špatně umístěná odpovědnost .....	301
O18: Nevhodný modifikátor static .....	301
O19: Používejte vysvětlující proměnné .....	301
O20: Názvy funkcí by měly sdělovat, co dělají.....	302
O21: Pochopte algoritmus .....	302
O22: Udělejte z logických závislostí fyzické .....	303
O23: Volte raději polymorfismus než příkazy if/else nebo switch/case .....	304
O24: Dodržujte standardní konvence .....	304
O25: Nahradte magická čísla pojmenovanými konstantami .....	305
O26: Buděte přesní .....	306
O27: Struktura je více než konvence .....	306
O28: Zapouzdřete podmínky .....	306
O29: Vyhýbejte se negativním podmíněným výrazům .....	306
O30: Funkce by měly provádět jen jednu věc .....	307
O31: Skrytá časová vazba .....	307
O32: Nepodléhejte libovůli.....	308
O33: Zapouzdřete hraniční podmínky.....	309
O34: Funkce by měly sestupovat jen o jednu úroveň abstrakce níže.....	309
O35: Mějte konfigurační data na vysokých úrovních.....	310
O36: Vyhnete se tranzitivním odkazům .....	311
<b>Java .....</b>	<b>311</b>
J1: Vyhnete se dlouhým seznamům a používejte zástupné znaky .....	311
J2: Vyhnete se dědění konstant .....	312
J3: Konstanty versus výčty .....	313
<b>Jména.....</b>	<b>314</b>
Jm1: Vybírejte popisná jména.....	314
Jm2: Vybírejte jména na adekvátní úrovni abstrakce .....	315
Jm3: Používejte standardní názvosloví všude, kde je to možné .....	316
Jm4: Jednoznačná jména .....	316
Jm5: Pro velké rozsahy používejte dlouhá jména .....	317
Jm6: Vyhnete se kódování jmen .....	317
Jm7: Jména by měla popisovat vedlejší efekty .....	317

---

Testy .....	318
T1: Nedostatečné testy .....	318
T2: Používejte nástroje pro pokrytí .....	318
T3: Nepřeskakujte triviální testy .....	318
T4: Opomenutý test je otázkou ohledně nejednoznačnosti .....	318
T5: Testujte hraniční podmínky .....	318
T6: V okolí programových chyb provádějte důkladné testy .....	318
T7: Zákonitosti v selhávání odhalují chyby .....	318
T8: Pokrytí kódu testy může odhalit chyby .....	319
T9: Testy by měly být rychlé .....	319
Závěr .....	319
Použitá literatura .....	319

## Dodatek A

### Souběžnost II ..... 321

Příklad klient/server .....	322
Server .....	322
Přidání podprocesů .....	323
Sledování serveru .....	324
Závěr .....	325
Počet cest při provádění kódu .....	325
Počet cest .....	326
Hlubší pohled .....	327
Závěr .....	329
Vyznejte se ve své knihovně .....	330
Běhový rámec .....	330
Řešení bez blokace .....	330
Bezpečné třídy bez podprocesů .....	332
Závislost mezi metodami může souběžný kód porušit .....	333
Tolerovat selhání .....	334
Zamykání na straně klienta .....	334
Zamykání na straně serveru .....	335
Zvyšování propustnosti .....	336
Kalkulace propustnosti jednoho podprocesu .....	337
Kalkulace propustnosti při více podprocesech .....	338
Zablokování .....	338
Vzájemné vyloučení .....	339
Zamkní a čekej .....	340
Zdroj nelze získat nucenou výměnou .....	340

Cyklické čekání .....	340
Řešení vzájemného vyloučení .....	340
Řešení problému „Zamkní a čekej“ .....	340
Řešení získávání zdrojů nucenou výměnou .....	341
Řešení cyklického čekání .....	341
Testování kódu s více podprocesy .....	342
Podpora nástrojů pro testování kódu s podprocesy .....	344
Závěr .....	345
Výukový program .....	345
Klient/server bez podprocesů .....	345
Klient/server s podprocesy .....	348

## Dodatek B

<b>org.jfree.date.SerialDate .....</b>	<b>351</b>
--	------------

Doslov .....	415
Rejstřík .....	417



# Předmluva

Jednou z oblíbených cukrovinek je u nás v Dánsku Ga-Jol, jejíž silné lékořicové aroma je výborným doplňkem našeho sychravého a mrázivého počasí. Kus šármu těchto cukrovinek spočívá pro nás Dány také v moudrých a duchaplných rčeních, vytíštěných v záhybu vrchní části každé krabičky. Dnes ráno jsem si koupil dvojité balení této lahůdky a našel jsem tam staré dánské pořekadlo:

Ærlighed i små ting er ikke nogen lille ting.

„Pocitost v malých věcech není malá věc.“ Bylo to dobré znamení, odpovídající tomu, co jsem chtěl říci již zde. Na malých věcech záleží. Tato kniha pojednává o prostých starostech, jejichž význam ale zdaleka malý není.

„Bůh spočívá v detailech,“ řekl architekt Ludwig mies van der Rohe. Tento citát připomíná současné argumenty ohledně role architektury ve vývoji softwaru a zvláště v oblasti, jako jsou „agilní techniky vývoje softwaru“. Čas od času Bob i já zjistujeme, že jsme do této bouřlivé debaty vtaženi. A také mies van der Rohe, který se zajímal o užitečnost a o nadčasové tvary staveb, což je základem dobré architektury. Na druhé straně také vybíral osobně každou kliku u dveří v domech, které navrhoval. Proč? Protože drobnosti jsou důležité. V naší pokračující „debatě“ na téma vývoje řízeného testy jsme Bob i já zjistili, že souhlasíme s tím, že architektura softwaru hraje v jeho vývoji důležitou roli, ačkoliv máme různé představy o tom, co to vlastně znamená. Relativně není takové slovíčkaření důležité, protože můžeme přijmout jako samozřejmost fakt, že zodpovědní profesionálové věnují *nějaký* čas přemýšlení a plánování hned na počátku projektu. Pojetí návrhu, vedeného pouhými testy a kódem z pozdních devadesátých let, je dálno za námi. Ale pozornost věnovaná detailům je stále nepostradatelnějším základem profesionalismu, než je jakákoli velkolepá vize. Za prvé je to praxe v malém, ve které profesionálové získávají zdatnost a důvěru pro praxi ve velkém. Za druhé, sebemenší kousek zanedbané konstrukce, dveří, které špatně doléhají, nebo křivá kachlička v podlaze, nebo dokonce jen nepořádek na stole dokáže zcela zmařit kouzlo celku. Právě o tom pojednává čistý kód.

Přesto je architektura jen jedním ze symbolů vývoje softwaru, což platí specificky pro tu jeho část, která vytváří počáteční *produkt* ve stejném smyslu, jako když architekt odevzdává bezchybnou budovu. V těchto dnech, patřících agilním metodám a metodám Scrum, je vše zaostřeno na rychlé uvedení *produkту* na trh. Chceme, aby továrna pracovala s maximální rychlosťí a produkovala software. Tohle jsou lidské továrny: myslíci a citliví kodéři, kteří zpracovávají nedostatky v produktu nebo sdělení uživatelů, na jejichž základě se má vytvořit *produkt*. V tomto pojetí se metafora výroby objevuje ještě silněji. Pohled na japonskou montážní automobilovou linku je velkou inspirací pro metodu Scrum.

Dokonce i v automobilovém průmyslu spočívá hlavní objem práce nikoliv ve výrobě, ale v údržbě – nebo v prevenci. V oblasti softwaru se 80 % toho, co děláme, nazývá „údržbou“: opravováním. Místo abychom se typicky západním způsobem soustředili na *produkci* dobrého softwaru, bychom měli myslet více jako domácí opravář ve stavebnictví nebo automechanik v automobilovém průmyslu. Co nám k *tomu* může říci japonský management?

Kolem roku 1951 se v Japonsku objevil nový přístup ke kvalitě výroby, nazvaný „absolutně produktivní údržba“ (Total Productive Maintenance – TPM). Zaměřuje se více na údržbu než na výrobu.

Jedním z hlavních pilířů TPM je několik tzv. zásad 5S. Zásady 5S jsou množinou pravidel – a zde používám termín „pravidlo“ z důvodu snazšího pochopení. Zásady 5S tvoří ve skutečnosti základy metody *zeštíhlení* – další otravný slogan na západní scéně, který tvrdě proniká do softwarových kruhů. Tyto principy nejsou volbou. Jak ukazuje strýček Bob na stránkách úvodu, praktiky tvorby dobrého softwaru vyžadují tato pravidla: pozornost, zachování rozvahy a přemýšlení. Nejde jen o to uvést zařízení továrny do optimální výrobní rychlosti. Filozofie 5S v sobě zahrnuje tyto myšlenky:

- ◆ *Seiri* – neboli organizace: Zásadní věcí je vědět, kde se věci nacházejí – základem je například vhodné pojmenování. Myslete si, že identifikátory proměnných nejsou důležité? Přečtete si o tom v následujících kapitolách.
- ◆ *Seiton* – neboli pořádek: Existuje jedno staré americké rčení: *A place for everything, and everything in its place* – Mějme pro každou věc místo a vše na svém místě. Úsek kódu by měl být tam, kde jej budete hledat – a pokud ne, měli byste ho refaktorovat, aby tam byl.
- ◆ *Seiso* – neboli čistění: Nemějte na pracovišti zavěšené dráty, olej, kovový odpad a smetí. Co zde říkají autoři o zaneřádění kódu poznámkami a vykomentovanými rádky, které zachycují historii nebo přání do budoucna? Zbavte se jich.
- ◆ *Seiketsu* – neboli standardizace: Skupina souhlasí s tím, jak si udržovat pracovní místo čisté. Myslete si, že tato kniha říká něco o konzistentním stylu kódování a sadě postupů v rámci skupiny? Kde se tyto standardy vzaly? Čtěte dále.
- ◆ *Shutsuke* – neboli disciplína. To znamená kázeň, dodržování postupů, věnovat často pozornost práci druhého a ochotu provádět změny.

Pokud přijmete výzvu – ano, výzvu – si přečtěte tento knihu a využívejte ji, oceněte tento poslední argument a porozumíte mu. Zde se konečně dostáváme ke kořenům odpovědného profesionálního přístupu v povolání, které by mělo být spojeno s životním cyklem produktu. V průběhu údržby automobilů nebo jiných strojů metodou TPM je ošetřování havarijních stavů – kdy chybou vyplynou na povrch – výjimkou. Místo toho jsme se dostali na jinou úroveň: kontrolujeme stroje každý den a opravujeme opotřebované součásti ještě než budou mít poruchu, tedy provedeme ekvivalent příslovečné výměny oleje po 15 000 kilometrech, abychom zabránili opotřebení a zadření. V případě kódu refaktorujte bez slitování. Můžete se ještě o jednu úroveň zlepšit, tak, jak inovovalo hnutí TPM před padesáti lety: na prvním místě konstruujte stroje, které se snadno udržují. Čitelnost vašeho kódu je stejně důležitá jako skutečnost, že se kód bude provádět. Poslední pravidlo, které bylo v kruzích TPM zavedeno kolem roku 1960, je zaměřit se na zavádění zcela nových strojů nebo na výměnu starých. Jak nás napomíná Fred Brooks, možná, že bychom měli předělat hlavní části kódu od samého počátku přiblížně každých sedm let, abychom vymetli špatně napsané partie. Možná bychom měli aktualizovat Brooksovou časovou konstantu řádově na týdny, dny nebo hodiny místo let. V tom spočívá ten rozdíl.

V detailu spočívá mnoho sil, nicméně je v tomto přístupu k životu něco skromného a hlubokého, jak bychom mohli stereotypně očekávat od čehokoliv, co má původ v Japonsku. Ale nejedná se jen o východní přístup k životu. Lidová moudrost je plná podobných ponaučení. Cítat v bodě *Seiton* vyšel z pera jednoho ministra státu Ohio, který doslově viděl upravenost „jako lék pro špatnost každého stupně“. A co *Seiso*? *Cistota je půl zdraví*. Při vši krásě, kterou budova oplývá, nepořádek v přijímací kanceláři ji bere veškerý týpyt. Co takhle *Shutsuke* v těchto malých záležitostech? *Kdo je věrný v malých věcech, je věrný i ve velkých*. Co žhavost po refaktorování v odpovídajících časech, posilující pozici pro následující „velká“ rozhodnutí, raději než jejich odklad? *Jeden včasný steh ušetří devět dalších. Ranní ptáče dál doskáče. Co můžeš udělat dnes, neodkládej na zítřek..* (To byl původní smysl

rčení „poslední důležitý okamžik“ v metodě zeštíhlení, dokud nepadlo do rukou softwarových konsultantů.) Co takhle nastavení prostoru pro malé, individuální snahy v rámci velkého celku? *Mohutný dub vyroste z malého žaludu.* Nebo co říkáte integraci jednoduché preventivní práce do každodenního života? *Gram prevence je lepší než tuna léků. S jedním jablkem denně nepotřebujete doktora.* Čistý kód spolu s pozorností vůči detailu ctí hluboké kořeny moudrosti, které se skrývají v naší obecné kultuře, nebo v její původní podobě nebo v podobě, v jaké by měla a může být.

Dokonce i v literatuře o velké architektuře můžeme nalézt rčení, která vycházejí z těchto domnělých detailů. Přemýšlejte o klikách miese van der Roheho. To je *seiri*. Znamená to věnovat pozornost každému jménu proměnné. Proměnnou byste měli pojmenovat se stejnou pečlivostí, s jakou dáváte jméno prvorozenému dítěti.

Jak jistě ví každý vlastník domu, podobná starost a neustálé vylepšování nikdy nekončí. Architekt Christopher Alexander – otec vzorů a jazyků vzorů – vidí v každém kroku vlastního návrhu malý, lokální čin opravy. A v dovednosti udělat dobrou stavbu vidí výhradně kompetenci architekta. Větší formy mohou být ponechány vzorům a jejich využití obyvatelům. Návrh je nekonečný nejen tím, že do domu přidáváme další pokoje, ale i tím, že věnujeme pozornost jejich vymalování, výměně opotrebovaných koberců nebo modernizaci kuchyňského dřezu. Většina dovedností vede k podobným názorům. V našem pátrání po jiných lidech, kteří věří, že boží dům se nachází v detailech, se ocitáme v dobré společnosti francouzského autora z 19. stolení Gustava Flauberta. Francouzský básník Paul Valery nám radí, že báseň není nikdy hotová a je třeba ji neustále předělávat. Když s tím přestanete, je to, jako kdybyste se jí zřekli. Takové zaujetí pro detaily je společné všem snahám o dokonalost. Možná, že tu najdete něco nového, ale možná že se během čtení této knihy rozhodnete přijmout pravidla, nad kterými již dávno zvítězila apatie nebo sklon k živelnosti, a prostě „zareagujete na změnu“.

Bohožel, podobné snahy běžně nechápeme jako základní kameny umění programovat. Kód brzy opouštíme ne proto, že je hotový, ale proto, že se náš hodnotový systém zaměřuje více na vnější vzhled než na podstatu toho, co poskytujeme. Cena za nedbalost se ukáže až na závěr: *Pravda vždy vyplave.* Ani v průmyslu, ani ve vědeckých kruzích se nikdo nesnižuje na takovou úroveň, aby se staral o čistý kód. Během mé minulé práce v Bellových laboratořích v oddělení výzkumu produkce softwaru (Software Production Research – skutečně produkce, výroba!) jsme přišli na několik stručných propočtů, které naznačovaly, že konzistentní způsob odsazování textu byl statisticky jedním z nejvýznamnějších ukazatelů nízkého počtu chyb. Chceme, aby architektura nebo programovací jazyk nebo nějaký jiný vyšší pojem byly záležitostí kvality. Tak, jako lidé, jejichž profesionnalismus má daleko do mistrovství nástrojů a grandiozních návrhových metod, i my cítíme pohanění důležitosti, kterou sem vnášejí tovární stroje, kodéři, za pomocí konzistentního používání odsazování. Abych citoval svou vlastní knihu, vydanou před 17 lety, takový styl odlišuje dokonalost od pouhé obratnosti. Japonský pohled na svět chápe zásadní význam každodenního pracovníka a k tomu i význam systému vývoje, jímž jsme zavázáni běžné a každodenní činnosti těchto pracovníků. Kvalita je výsledkem milionu obětavých kroků péče – nejen nějaké grandiozní metody, která sestoupila z nebe. Že jsou tyto kroky jednoduché, to ještě neznamená, že jsou zjednodušující, a už vůbec ne, že by byly snadné. Nicméně jsou základní kostrou dokonalosti a také krásy v rámci každého lidského snažení. Ignorovat je znamená nebýt plně lidským.

Samozřejmě že jsem stále obhájem myšlení v širším slova smyslu a zvláště obhájem hodnoty architektonického přístupu, hluboce zakořeněného v oblasti znalostí a softwarové upotřebitelnosti. O tom tato kniha není – alespoň ne zjevně. Tato kniha obsahuje delikátnější sdělení, jehož hloubka by neměla být podceněna. Odpovídá aktuálnímu současnemu rčení lidí, jako jsou Peter Sommerlad, Kevlin

Henney a Giovanni Asproní, kteří se kódům věnují na plný úvazek: Jejich mantrou je „kód je návrhem“ a „jednoduchý kód“. Zatímco my musíme mít na paměti, že rozhraní je program a že jeho struktury mají co říci k jeho struktuře, zásadní je natrvalo přijmout jednoduchý postoj, že návrh žije v kódě dál. A zatímco předělávání v přeneseném smyslu výroby vede k nákladům, předělávání v oblasti návrhu vede k vyšším hodnotám. Na náš kód bychom měli pohlížet jako na skvělé vyjádření ušlechtilého úsilí vytvořit návrh – návrh jako proces, ne jako statický cíl. Je to kód, kde se uplatní metrika vazeb a soudržnosti v architektuře. Pokud budete poslouchat, jak Larry Constantine popisuje vazby a soudržnost, mluví v termínech kódů – ne v abstraktních a velkolepých koncepcích, které lze nalézt v UML. Richard Gabriel nám ve své eseji „Abstraction Descant“ (chvála abstrakce) tvrdí, že abstrakce pochází od dábla. Kód je proti dáblu a čistý kód je snad božský.

Když se vrátím zpět ke své malé krabičce Ga-Jol, myslím, že je důležité si všimnout, že dánská moudrost nám radí, abychom věnovali pozornost nejen malým věcem, ale také abychom v malých věcech byli poctiví. To znamená být poctivý ke kódům, poctivý ke svým kolegům ohledně stavu kódů a především být poctivý v otázce kódů vůči sobě. Udělali jsme *to nejlepší*, abychom „zanechali tábořiště čistější, než bylo předtím“? Refaktorovali jsme svůj kód před začátkem práce? To nejsou okrajové záležitosti, ale věci, které leží přímo v ohnisku hodnot agilních metod. Doporučený postup v metodě Scrum je, aby refaktorování bylo součástí koncepce fáze „hotovo“. Ani architektura, ani čistý kód netrvá na perfektnosti, ale jen na poctivosti a nejkvalitnější práci, které jsme schopni. *Chybovat je lidské, odpouštět božské*. V rámci metody Scrum děláme vše viditelné. Větráme naše špinavé prádlo. Jsme poctiví ohledně stavu našeho kódů, protože ten není nikdy perfektní. Staneme se lidštějšími, hodnotnějšími v dokonalosti a staneme blíže této velikosti v detailech.

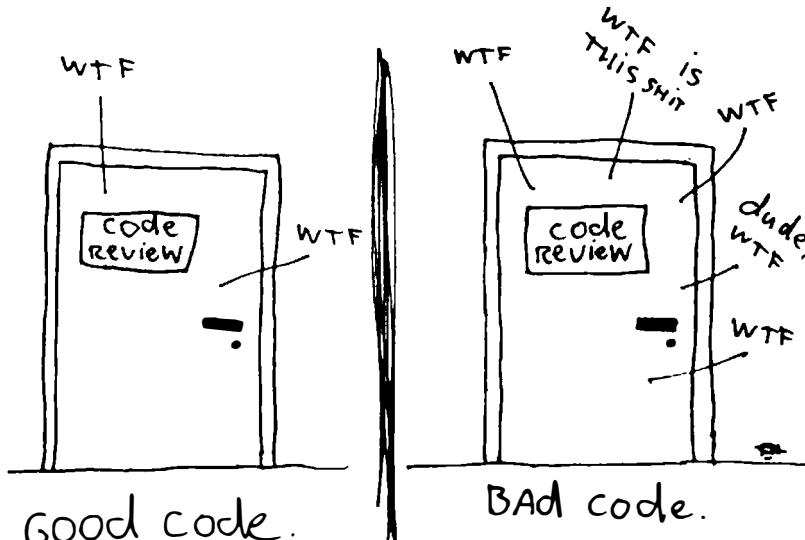
V naší profesi zoufale potřebujeme veškerou pomoc, kterou můžeme získat. Pokud čistá podlaha sníží v obchodě počet úrazů a pokud dobré zorganizované pracovní prostředky zvýší v dílně produktivitu, pak jsem pro ně. Pokud jde o tuto knihu, jde o nejlepší praktickou aplikaci zásad „zeštílení“ v oblasti softwaru, kterou jsem kdy viděl v tisku. Od této malé a praktické skupiny přemýšlejících intelektuálů, která po léta spolupracovala nejen na tom, aby se zlepšili, ale také aby obdarovali svými vědomostmi lidi z průmyslové sféry dílem, které právě držíte v rukách, jsem nečekal nic menšího. Svět to dělá o něco lepším než jsem jej viděl dříve, dokud mi strýček Bob neposlal svůj rukopis. Když jsem skončil tohle cvičení v ušlechtilém pronikání do podstaty věci, mohu odejít, abych si uklidil svůj psací stůl.

**James O. Coplien**

Mørdrup, Denmark

# Úvod

Jediná skutečná míra kvality kódu:  
Počet průšvihů za minutu



Reprodukované s laskavým svolením Thoma Holwerdy. [http://www.osnews.com/story/19266/WTFs\\_m](http://www.osnews.com/story/19266/WTFs_m)  
(c) 2008 Focus Shift

Do kterých dveří patří váš kód? Kde se nachází váš tým a vaše společnost? Proč jsme v této místnosti? Jde o obyčejné přezkoumání kódu nebo jsme hned po startu nalezli moře příšerných problémů? Ladíme v panice a zíráme do kódu, o kterém jsme si mysleli, že funguje? Opouštěj nás hromadně zákazníci a manažeři nám dýchají na záda? Jak si můžeme být jisti, že skončíme za pravými dveřmi, když postup začíná být obtížný? Odpověď zní: *mistrovská práce*.

Cesta k mistrovství má dvě části: znalosti a práci. Musíte získat znalosti principů, vzorů, postupů a heuristiky, kterou mistr v oboru ovládá, a musíte tuto znalost vstřebat do svých prstů, očí a do své osobnosti tím, že budete tvrdě pracovat a cvičit.

Mohu vás naučit fyziku řízení bicyklu. Opravdu, klasická matematika je relativně jednoduchá. Přitažlivost, tření, moment hybnosti, těžiště atd. lze všechno ukázat na méně než jedné stránce popsané rovnicemi. S těmito vzorečky mohu dokázat, že jízda na kole je praktická, a poskytnout vám veškerou znalost, kterou potřebujete, abyste jezdili. A přesto spadnete na zem ihned, jakmile si na kolo sednete.

Psaní kódu je úplně stejné. Mohli bychom sepsat všechny zásady správného postupu při psaní čistého kódu a pak vám tuto práci svěřit (jinými slovy, nechat vás spadnout z kola ihned, jakmile na něj nasednete), ale jaké by to z nás dělalo učitele a jaké žáky z vás?

Ne. Takhle tato kniha postavena není.

Učit se psát čistý kód je *dřina*. Vyžaduje to víc než jen znalost principů a vzorů. Musíte se nad tím *potit*. Musíte to sami procvičovat a sledovat své neúspěchy. Musíte sledovat pokusy ostatních i jejich neúspěchy. Musíte je vidět klopýtat a vracet se zpět. Musíte je vidět lámat si hlavu nad svými rozodnutími a vidět cenu, kterou platí za rozhodnutí, jež jsou chybňá.

Buděte připraveni na to, že čtení této knihy bude dřina. Tohle není kniha pro dobrou pohodu, kterou můžete číst na palubě letadla a již skončíte před přistáním. Tato kniha vás přinutí pracovat, *a to tvrdě*. Jaký druh práce budete vykonávat? Budete čist kód – mnoho kódu. A budete vedeni k přemýšlení, co je na tomto kódu dobrého a co špatného. Budete žádání, abyste sledovali, jak jednotlivé moduly rozebráme a pak je opět dáváme dohromady. To zabere čas i úsilí, ale myslíme si, že se to vyplatí.

Rozdělili jsme tuto knihu na tři části. Několik prvních kapitol popisuje principy, vzory a postupy, jak psát čistý kód. V těchto kapitolách je poměrně dost kódu, který vás bude lákat ke čtení. Zde budete připraveni pro druhou část, která bude následovat. Pokud tuto knihu po přečtení první části odložíte, pak hodně štěstí!

Druhá část knihy je těžší. Skládá se z několika rozborů problémů, jejichž složitost postupně narůstá. Každý rozbor je cvičením v čistění nějakého kódu – transformování problematického kódu do jiného tvaru, který tolik problémů nemá. V této sekci se intenzivně věnujeme detailům. Budete muset lisovat sem a tam mezi doprovodným textem a výpisem kódu. Budete muset analyzovat kód, se kterým pracujeme, porozumět mu a projít si naši argumentaci pro každou změnu, již provedeme. Rezervujte si na to nějaký čas, protože *tohle by vám mělo trvat několik dnů*.

Třetí část knihy je za odměnu. Je to jediná kapitola obsahující seznam heuristik a intuitivních postupů, sebraných v průběhu tvorby našich rozborů. Když jsme tyto případy procházeli a čistili jejich kód, zdokumentovali jsme veškeré důvody svých zásahů buď jako heuristiku nebo intuitivní postup. Pokusili jsme se porozumět svým vlastním reakcím na kód, který jsme četli a opravovali, tvrdě jsme pracovali a snažili se podchytit, proč jsme se cítili právě takto a proč jsme dělali to, co jsme dělali. Výsledkem je databáze znalostí, která popisuje způsob našeho myšlení při psaní, čtení a čistění kódu.

Tato databáze znalostí má omezenou hodnotu, pokud si nedáte práci a nebudeste pečlivě studovat rozitory ve druhé části této knihy. V těchto studiích jsme pečlivě popsali každou změnu, kterou jsme provedli, pomocí dopředných odkazů na heuristiku. Tyto odkazy jsou umístěny v hranatých závorkách, např. [H22]. To vám umožní vidět souvislosti, jak byly tyto heuristiky použity a napsány. Nejsou to samotné heuristiky, které jsou tak cenné, ale *souvislost mezi těmito heuristikami a samostatnými rozhodnutími, jež jsme během čistění kódu v našich rozborech udělali*.

Abychom vám s těmito souvislostmi pomohli, umístili jsme na konec knihy křížové odkazy, které pro každý dopředný odkaz uvádějí příslušné číslo stránky. Takto lze najít všechna místa, kde byla nějaká konkrétní heuristika použita.

Pokud si přečtete první a třetí sekci a přeskočíte rozitory problémů, pak jste četli další „oddechovu“ knihu o psaní dobrého softwaru. Ale pokud věnujete čas a propracujete se těmito studiemi krok za krokem a s každým krátkodobým rozhodnutím, pokud to vezmete z našeho pohledu a přinutíte

se přemýšlet tím stejným způsobem jako my, získáte mnohem bohatší pochopení těchto principů, modelů, postupů a heuristik. Pak už to nikdy nebude nějaká „oddechová“ znalost. Bude to něco, co jste dostali do krve, do prstů a do srdce. Bude to něco, co se stalo vaši součástí stejným způsobem, jako se kolo stane součástí vašeho já, jakmile zvládnete jízdu na něm.

## Poděkování

### Výtvarné dílo

Děkuji mým dvěma umělkyním, Jeniffer Kohnke a Angele Brooks. Jennifer se zasloužila o senzační a originální obrázky na začátku každé kapitoly a rovněž o portréty lidí, jako jsou Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers a já.

Angela se zasloužila o znamenité obrázky, které zdobí vnitřní části každé z kapitol. Také pro mě v několika minulých letech namalovala němalé množství obrázků, včetně těch, které se nacházejí uvnitř knihy „*Agile Software Development: Principles, Patterns, and Practices*“. Je také mým prvorozeným dítětem, ze kterého mám velkou radost.

## Poznámka redakce českého vydání

Nakladatelství Computer Press, které pro vás tuto knihu přeložilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

Computer Press  
redakce PC literatury  
Holandská 8  
639 00 Brno

nebo

[knihy@cpress.cz](mailto:knihy@cpress.cz)

Další informace a případné opravy českého vydání knihy najdete na internetové adrese <http://knihy.cpress.cz/k1646>. Prostřednictvím uvedené adresy můžete též naši redakci zaslat komentář nebo dotaz týkající se knihy. Na vaše reakce se srdečně těšíme.



# KAPITOLA 1

## Čistý kód

### V této kapitole najdete:

- ◆ Kód nezanikne
- ◆ Špatný kód
- ◆ Celková cena za nepořádek
- ◆ Celková rekonstrukce na zelené louce
- ◆ Přístup
- ◆ Prvotní paradox
- ◆ Umění čistého kódu?
- ◆ Co je to čistý kód?
- ◆ Myšlenkový směr
- ◆ Autoři
- ◆ Skautské pravidlo
- ◆ Úvod a principy
- ◆ Závěr



Tuto knihu čtete ze dvou důvodů. Za prvé proto, že programujete. Za druhé proto, že chcete programovat lépe. Dobrá. Potřebujeme lepší programátory.

Tato kniha pojednává o dobrém programování. Je plná kódu. Budeme jej zkoumat ze všech možných úhlů. Podíváme se na něj odshora dolů, zdola nahoru a skrze něj zevnitř ven. Až skončíme, budeme toho o kódu dost vědět. Navíc budeme schopni říci, jaký je rozdíl mezi dobrým a špatným kódem. Budeme vědět, jak psát dobrý kód. A budeme vědět, jak předělat špatný kód na dobrý.

## Kód nezanikne

Někdo by si mohl myslit, že kniha o kódu je něco zastaralého – že toto téma již není aktuální. Že bychom se měli místo toho zabývat spíše modely a požadavky na ně. Někteří lidé skutečně prohlašovali, že se nacházíme blízko okamžiku, kdy kód přestane být významný. Že brzy bude veškerý kód generován a nebude ho psát. Že ho programátoři nebudou jednoduše potřebovat, protože podnikatelé budou generovat programy podle svých specifikací.

Nesmysl! Kódu se nikdy nezbavíme, protože zahrnuje detaily požadavků. Na určité úrovni nelze tyto detaily ignorovat nebo od nich abstrahovat. Je nutné je specifikovat. A specifikace požadavků v takovém detailu, aby je počítáč mohl provést, je *programování*. Takovou specifikaci je kód.

Předpokládám, že úroveň abstrakce jazyků bude stále vzrůstat. Rovněž předpokládám, že počet doménově specifických jazyků stále poroste. Bude to dobrý krok. Ale to nebude znamenat konec kódu. Ve skutečnosti *budou* všechny specifikace v těchto jazyčích vyšší úrovni nebo doménově specifických jazyčích kódem! Ten bude muset být stále precizní, přesný a natolik formální a detailní, aby mu počítáč dokázal rozumět a aby ho mohl provést.

Lidé, kteří si myslí, že jednoho dne zmizí, jsou jako matematici, již doufají, že jednoho dne objeví matematiku, která nemusí být formální. Doufají, že jednoho dne objeví způsob, jak vytvořit stroje, které budou umět provést to, co chceme, nikoli to, co říkáme. Tyto stroje nám budou muset rozumět tak dobře, že budou umět přeložit nejasně formulované potřeby do bezchybně fungujících programů, které přesně těmto potřebám vyhoví.

K tomu nikdy nedojde. Dokonce ani lidé s veškerou svou intuicí a tvůrčími schopnostmi nebyli schopni vytvořit úspěšné systémy na základě nejasných pocitů svých zákazníků. Opravdu, pokud nás něco naučilo pravidlům pro specifikaci požadavků, pak to byl fakt, že dobré specifikované požadavky jsou formální právě tak, jako je formální kód, a mohou fungovat jako spustitelný test tohoto kódu!

Nezapomínejte, že kód je skutečně jazykem, ve kterém požadavky definitivně vyjadřujeme. Můžeme vytvořit jazyky, které jsou takovým požadavkům blíže. Můžeme vytvořit nástroje, které nám umožní tyto požadavky rozložit a opět složit do formálních struktur. Ale nikdy neodstraníme nezbytnou přesnost – kód tedy nikdy nezanikne.

## Špatný kód

Nedávno jsem četl úvod ke knize *Implementation Patterns*<sup>1</sup>, jejímž autorem Kent Beck.

Říká: „...tato kniha je založena na dost křehkém předpokladu: že na dobrém kódu záleží...“ Křehký předpoklad? Nesouhlasím! Myslím, že tento předpoklad je jeden z nejsilnějších, podporovanějších

1. [Beck07].

a nejpřetíženějších předpokladů v našem oboru (a myslím si, že Kent to ví). Víme, že dobrý kód má svůj význam, protože jsme se dlouho museli potýkat s jeho nedostatkem.

Znám jednu společnost, která v druhé polovině osmdesátých let napsala aplikaci – trhák. Aplikace byla velmi populární, mnoho profesionálů si ji koupilo a používalo ji. Ale pak se cykly aktualizací začaly prodlužovat. Chyby mezi jednotlivými aktualizacemi nebyly opraveny. Čas načtení do paměti rostl a počet havárií také. Vzpomínám si na den, kdy jsem díky naprosté frustraci program ukončil a již jsem jej nikdy nepoužil. Krátce na to společnost s podnikáním skončila.



O dvě desetiletí později jsem potkal jednoho z prvních zaměstnanců této společnosti a zeptal jsem se ho, co se stalo. Jeho odpověď potvrdila moje obavy. Spěchali s produktem na trh a zanechali v kódu značný nepořádek. Přidávali další a další vlastnosti a kód ztrácel na kvalitě až do okamžiku, kdy to již dále nemohli zvládnout. *Tuto společnost položil špatný kód.*

Stalo se někdy i vám, že by vás zdržoval špatný kód? Jestliže programujete a máte nějaké zkušenosti, museli jste podobné překážky pocítit mnohokrát. Opravdu, existuje pro to jméno. Nazýváme to *prokousávání* (wading). Špatným kódem se prokousáváme. Pachtíme se bažinou spletitého ostružíni a skrytých pastí. Usilujeme o to, abychom si našli cestu, doufáme, že nalezneme nějaké vodítko, nějakou stopu toho, o co tu jde. Ale vše, co vidíme, je stále nesmyslnější kód.

Samozřejmě že vás někdy trápil špatný kód. Takže – proč jste ho napsali?

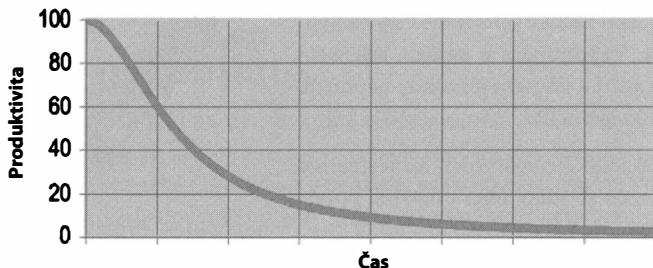
Zkoušeli jste psát příliš rychle? Pracovali jste ve spěchu? Pravděpodobně ano. Možná jste cítili, že nemáte čas na pořádnou práci, že by se na vás šéf naštval, kdybyste si udělali čas a kód si vyčistili. Snad jste byli pouze unaveni z práce na tomto programu a chtěli jste to mít za sebou. Nebo jste se možná podívali na nějaké jiné resty, které jste slíbili dokončit, a zjistili jste, že potřebujete dát tento modul dohromady, abyste se mohli pustit do dalšího. Něčím podobným jsme prošli všichni.

Nám všem se někdy stalo, že jsme se podívali na nepořádek, který jsme právě napáchali, a pak jsme si řekli, že to necháme na další den. Všichni jsme si prožili úlevu, když jsme viděli, jak náš zanezáděný program funguje, a rozhodli jsme se, že fungující binec je lepší než nic. Všichni jsme si řekli, že se k tomu ještě vrátíme a vyčistíme to později. Samozřejmě jsme tehdy neznali LeBlancův zákon: *Později znamená nikdy.*

## Celková cena za nepořádek

Pokud jste programovali více než dva nebo tři roky, pravděpodobně vaši práci významně zpomalil zanezáděný kód někoho jiného. Míra takového zpomalení může být značná. V rozpětí roku nebo dvou mohou týmy, které na začátku pracovaly velmi rychle, náhle zjistit, že se pohybují hlemýždím temtem. Každá změna kódu, kterou provedou, způsobí nefunkčnost dvou nebo tří jiných částí kódu. Žádná změna není triviální. Každé přidání nebo každá změna systému vyžaduje pochopit jeho spletitosti, fígle a kličky tak, aby bylo možné přidat další spletitosti, fígle nebo kličky. Časem bude míra nepořádku tak rozsáhlá, hluboká a vysoká, že to nikdo nedokáže vyčistit. Není to prostě možné.

Jak se nepořádek vrší, produktivita týmu se postupně snižuje a asymptoticky se blíží k nule. S klesající produktivitou udělá vedení jedinou věc, kterou umí: rozšíří počet zaměstnanců projektu s nadějí na její zvýšení. Ale noví zaměstnanci se v návrhu systému nevyznají. Neznají rozdíl mezi změnami, které jsou v souladu se záměrem návrhu, a změnami, jež jsou s ním v rozporu. Navíc jsou tito lidé, podobně jako všichni ostatní v týmu, pod strašným tlakem zvýšit produktivitu. Takže vytvářejí další a další nepořádek a stahují produktivitu dále směrem k nule. (Viz obrázek 1.1.)



Obrázek 1.1. Závislost produktivity na čase

## Celková rekonstrukce na zelené louce

Nakonec se tým vzbouří. Sdělí vedení, že s vývojem tak hrozného aplikačního kódu nemohou dálé pokračovat. Vyžadují nový návrh. Vedení nechce vynakládat zdroje na zcela nový návrh projektu, ale nemůže poprít, že produktivita práce je příšerná. Nakonec couvnou před požadavky vývojářů a k celkové rekonstrukci dají souhlas.

Je vybrán nový elitní tým. V něm by chtěl být každý, protože se jedná o projekt na zelené louce. Začnou pracovat a vytvářet něco opravdu obdivuhodného. Pro tento tým jsou vybráni jen ti nejlepší a nejchytřejší. Všichni ostatní musí pokračovat v údržbě současného systému.

Nyní spolu závodí dva týmy. Elitní tým musí vybudovat nový systém, který umí všechno, co zvládá ten starý. Nejen to, ale musí držet krok s veškerými změnami, které se kontinuálně provádějí na starém systému. Vedení nahradí starý systém teprve tehdy, když ten nový bude umět vše, co umí starý.

Tento závod může trvat dost dlouho. Znám případ, kdy to trvalo 10 let. A v okamžiku, kdy je nový systém hotový, původní členové elitního týmu jsou již dálno pryč a současní členové požadují nový návrh systému, protože je to jeden veliký nepořádek.

Pokud máte zkušenosť jen s částí tohoto vyprávění, tak již víte, že strávit čas na údržbě čistého kódu je nejen cenově efektivní, ale je to záležitost profesionálního přežití.

## Přístup

Už jste někdy plavali v zaneřáděném kódu tak, že vám zabralo týden to, co mělo trvat hodiny? Už jste viděli, že to, co mělo být změnou na jednom rádku, znamenalo změny ve stovkách různých modulů? Všechny tyto symptomy jsou velmi časté.

Proč se to kódu stává? Proč dobrý kód degeneruje tak rychle a stane se z něj špatný kód? Máme pro to mnoho vysvětlení. Stěžujeme si, že se požadavky změnily tak, že původnímu návrhu odporuji.

Pláčeme nad plány, které byly natolik přísné, že práci nemůžeme udělat dobře. Plácáme o pitomých manažerech, netolerantních zákaznících, nepoužitelných týpcích z marketingu a telefonních zlepšováčích. Ale chyba, milý Watsone, není ve hvězdách, ale v nás. Nejsme profesionálové.

Tohle může být docela hořká pilulka. Jak může být tento nepořádek *naší* chybou? Co požadavky? Co časový rozvrh? Co hloupí manažeři a nepoužitelní týpci z marketingu? Nemají také část viny?

Ne. Manažeři a pracovníci v marketingu hledají *u nás* informace, které potřebují, aby mohli něco slíbit a přijmout závazky, a i když to třeba neočekávají od nás, proč bychom měli být nesmělí a neříct jim, co si myslíme? Uživatelé od nás očekávají, že jim potvrďme možnosti, jak by se mohly jejich požadavky do systému zavést. Projektoví manažeři od nás očekávají, že jim pomůžeme vypracovat plán práce. Máme velký podíl spoluviny na plánování projektu a sdílíme značnou část odpovědnosti za jakékoli selhání, zvláště pokud jsou tato selhání způsobena špatným kódem!

„Ale počkejte!“ řeknete si. „Pokud neudělám to, co mi řekne manažer, dostanu vyhazov.“ Pravděpodobně ne. Většina manažerů chce znát pravdu, i když tak často nejednají. Většina manažerů chce dobrý kód, i když jsou posedlí dodržováním rozvrhu práce. Mohou třeba zaníceně bránit časový rozvrh a požadavky, ale to je jejich práce. Vaše práce je stejně vášnivě obhájit kód.

Abychom tento bod uzavřeli, co kdyby vás jako doktora žádal váš pacient, abyste přestali s tím hloučkou mytí rukou, když se připravujete na operaci, protože to zabírá příliš mnoho času?<sup>2</sup>

Je jasné, že šéfem je pacient a přesto by měl lékař tento požadavek rozhodně odmítout. Proč? Protože lékař ví o riziku nemoci a infekce více než pacient. Bylo by neprofesionální (nebo i zločinné), kdyby lékař pacientovi vyhověl.

Stejně tak je neprofesionální od programátora, aby ustoupil požadavku manažera, který nerozumí riziku zavádění nepořádku do kódu.

## Prvotní paradox

Programátoři musejí čelit paradoxu základních hodnot. Všichni vývojáři s víceletou zkušenosí vědí, že předchozí nepořádek jejich práci zpomaluje. A přesto všichni vývojáři cítí tlak, který je vede k nepořádnému kódu, aby vyhověli konečným termínům. Stručně řečeno, nevěnují dost času tomu, aby pracovali rychle.

Skuteční profesionálové vědí, že druhá část paradoxu je špatně. Nevyhovíte termínu tím, že vytvoříte špatný kód. Opravdu, nepořádek v kódu vás okamžitě zpomalí a znemožní vám splnění termínu. *Jedinou* možností, jak stihnout termín – jedinou možností, jak postupovat rychle – je neustále udržovat kód tak čistý, jak jen to je možné.

## Umění čistého kódu?

Řekněme, že věříte, že zaneřáděný kód je závažnou překážkou. Řekněme, že akceptujete, že jedinou možností, jak pracovat rychle, je udržovat svůj kód čistý. Pak se musíte ptát sami sebe: „Jak mám psát čistý kód?“ Není dobré zkoušet psát čistý kód, když nevíte, co to znamená!

2. Když v roce 1847 poprvé doporučil Ignác Semmelweis lékařům mytí rukou, byl odmítnut na základě toho, že lékaři byli příliš zaneprázdněni a mezi jednotlivými návštěvami pacientů neměli na mytí rukou čas.

Špatnou zprávou je, že psaní čistého kódu se hodně podobá malování obrazu. Většina z nás pozná, kdy je obraz namalován dobře a kdy špatně. Ale schopnost rozeznat dobré umění od špatného neznamená, že víme, jak malovat. Takže schopnost rozpoznat čistý kód od nečistého neznamená, že víme, jak jej máme psát!

Psaní čistého kódu vyžaduje ukázněné používání nesčetných drobných technik, používaných za pomocí pečlivě získaného smyslu pro „čistotu“. Tento „smysl pro kód“ je klíčovou záležitostí. Některí z nás se s ním narodí. Některí z nás se musí snažit, aby jej získali. Nejen že nám umožní vidět, zda je kód dobrý či špatný, ale ukazuje nám i strategii, jak použít náš výcvik k transformaci špatného kódu na čistý.

Programátor bez tohoto „smyslu pro kód“ se může dívat na zaneřáděný modul a nepořádek rozeznat, jenže nebude mít ponětí, co s tím má dělat. Programátor se „smyslem pro kód“ se podívá na zaneřáděný modul a uvidí různé volby a alternativy. „Smysl pro kód“ pomůže programátorovi volit tu nejlepší alternativu a povede ho nebo ji v plánování posloupnosti změn, které zachovají funkčnost a dostat se z jednoho místa na druhé.

Stručně řečeno, programátor píšící čistý kód je umělcem, který začne s čistým plátnem a během série transformací skončí u systému s elegantním kódem.

## Co je to čistý kód?

Definicí čistého kódu je pravděpodobně tolik, kolik je programátorů. Takže jsem se zeptal několika velmi známých a velmi zkušených programátorů, co si o tom myslí.

### Bjarne Stroustrup, tvůrce jazyka C++ a autor knihy „The C++ Programming Language“

*Chci, aby můj kód byl elegantní a účinný. Logika by měla být přímočará, aby se chyby neměly kde skrývat, s minimálnimi závislostmi, aby údržba byla jednoduchá, kompletní ošetření chyb v souladu s jasně formulovanou strategií a s výkonem blízkým optimu, aby lidé nebyli v pokusu zavádět do kódu nepořádek pomocí svévolných optimalizací. Čistý kód plní svou funkci dobře.*

Bjarne používá slovo „elegantní“. To je slovo! Slovník v MacBook® uvádí následující definice: *příjemně ladný a módní ve vzhledu nebo způsobu; příjemně důmyslný a jednoduchý*. Všimněte si důrazu na slovo „*příjemný*“. Bjarne je očividně toho názoru, že čistý kód je *příjemně čist*. Čtení čistého kódu by u vás mělo vyvolat příjemné naladění tak, jako dovedně zhotovená hudební skříň nebo dobré navržené auto.



Bjarne zmiňuje také účinnost – *dvakrát* To by nás možná nemělo u tvůrce jazyka C++ překvapovat, ale já si myslím, že je v tom skryto více, než pouhý požadavek na rychlosť. Zbytečné cykly elegantní nejsou, nejsou potěšující. A nyní si všimněte, jaké slovo používá Bjarne k popisu důsledků této nelegantnosti. Používá slovo „*pokušení*“. V tomto je hluboká pravda. Špatný kód uvádí nepořádek v pokusu, aby rostl! Pokud provádějí změny ve špatném kódu jiní lidé, mají tendenci ho ještě zhoršit.

Pragmatický Dave Thomas a Andy Hunt říkají totéž, ale jiným způsobem. Použili metaforu vyraženého okna<sup>3</sup>.

Budova s vyraženými okny vypadá, jako by se o ni nikdo nestaral. Takže se o ni lidé přestanou starat. Umožní, že někdo vyrazí další okna. Nakonec je sami budou aktivně vyrážet. Fasádu poničí sprejem a budou tolerovat hromadění odpadků. Jedno vyražené okno bude počátkem procesu rozkladu.

Bjarne rovněž uvádí, že zpracování chyb by mělo být kompletní. Zde se dostáváme do oblasti pravidel pro zaměřování se na detaily. Zkrácené zpracování chyb je jen jedním ze způsobů, jakým programátoři detaily odbývají. Dalším problémem je nevracení paměti, jiným zase souběh událostí, jiným nekonzistentní tvorba jmen. Výsledkem je, že čistý kód je výrazem velké péče o detail.

Bjarne v závěru ujišťuje, že čistý kód dělá svou práci dobře. Není špatné, že existuje tolik principů pro navrhování kódu, které je možné zredukovat na toto jednoduché upozornění. Různí autoři se pokoušeli jeden za druhým tuto myšlenku vysvětlit. Špatný kód se toho snaží dělat příliš mnoho, má zmatený záměr a mnohoznačný smysl. Čistý kód je *jasně zaměřen*. Každá funkce, každá třída, každý modul vyjadřuje soustředěný přístup, který zůstává neznečistěný a vlivem okolních detailů se vůbec nerozptyluje.

### **Grady Booch, autor knihy „Object Oriented Analysis and Design with Applications“**

*Čistý kód je jednoduchý a přímočarý. Čistý kód se čte jako dobré napsaná próza. Čistý kód nikdy nezatemňuje záměr návrháře, ale je plný břitkých abstrakcí a přímých toků řízení.*

Grady je v něčem zajedno s Bjarnem, ale dívá se na věc z pohledu *čitelnosti*. Velmi se mi líbí jeho názor, že čistý kód bychom měli číst jako dobré napsanou prózu. Zkuste si vybavit nějakou dobrou knihu, kterou jste četli. Vzpomeňte si, jak v ní mizela slova, která nahradila obrázky! Jako byste sledovali film, nebo ne? Výstižněj! Viděli jste postavy, slyšeli jste zvuky, vnímali jste patos a humor.



Čtení čistého kódu nebude nikdy úplně stejně jako čtení *Pána prstem*. Přesto není literární metafora tak úplně špatná. Jako v případě dobrého románu, čistý kód by měl jasně odhalit napětí v problému, který má řešit. Tohle napětí by měl stupňovat až ke klimaxu a pak čtenáři sdělit, že „Aha! Samozřejmě!“, a s odhalením samozřejmého řešení mizí jak problémy, tak i napětí.

Myslím si, že Gradyho obrat „břitká abstrakce“ je fascinující protimluv! Koneckonců je slovo „břitký“ blízko slova „konkrétní“. Můj slovník MacBook uvádí následující definici „břitký“: *věcně přesvědčivý a skutečný, bez rozpáku a zbytcích detailů*. Navzdory tomuto zdánlivému významovému protikladu v sobě tato slova skrývají důležitou myšlenku. Náš kód by měl být skutečný v protikladu ke spekulativnímu. Měl by obsahovat jen to, co je nezbytné. Naši čtenáři by nás měli chápát jako rozhodné.

3. <http://www.pragmaticprogrammer.com/booksellers/2004-12.html>

## „Velký“ Dave Thomas, zakladatel OTI, kmotr strategie Eclipse

Čistý kód je čitelný a může jej vylepšovat i jiný vývojář než jeho původní autor. Má jednotkové a akceptační testy. Používá smysluplná jména. Umožňuje raději jeden způsob provedení nějakého úkolu než několik. Obsahuje minimální počet závislostí, které jsou jasné definovány. Používá API srozumitelně a v minimální míře. Kód by měl být kultivovaný (t.j. opatřen dokumentačními poznámkami, pozn. překl.), protože v závislosti na jazyku nelze všechny nezbytné informace vyjádřit v samotném kódu.

Velký Dave sdílí Gradyho přání čitelnosti, ale je zde důležitý obrat. Dave tvrdí, že čistý kód umožňuje *ostatním*, aby jej mohli jednoduše vylepšovat. To se může zdát samozrejmé, ale nelze to přeceňovat. Koneckonců, existuje rozdíl mezi kódem, který lze jednoduše čist, a kódem, jejž lze jednoduše měnit.



Dave spojuje čistotu s testy! Před deseti lety by tohle způsobilo mnoho překvapení. Ale metoda TDD (Test Driven Development, vývoj řízený testy) měla na naše odvětví hluboký dopad a stala se jedním z našich nejfundamentálnějších disciplín. Dave má pravdu. Kód bez testů není čistý. Je jedno, jak je elegantní, je jedno, jak je čitelný a přístupný. Nemá-li testy, je nečistý.

Dave používá dvakrát slovo *minimální*. Očividně si cení více kódu, který je malý, než kód, jenž je rozsáhlý. Skutečně, tohle byl obvyklý refrén v softwarové literatuře od jejího počátku. Menší je lepší.

Dave rovněž říká, že kód by měl být *kultivovaný*. To je jemná narážka na *kultivované programování*<sup>4</sup>.

Celkový výsledek je, že kód by se měl vytvářet v takové podobě, aby jej lidé mohli čist.

## Michael Feathers, autor knihy „Working Effectively with Legacy Code“

Mohl bych vyjmenovat veškeré dobré vlastnosti, kterých jsem si u čistého kódu všiml, ale jedna z nich je zastřešující a vede ke všem ostatním. Čistý kód vypadá vždy tak, jako by ho psal někdo pečlivý. Neexistuje nic samozrejmého, co můžete udělat pro jeho zlepšení. O všech těchto záležitostech již autor kódu přemýšlel a jestliže se pokusíte uvažovat o nějakých vylepšeních, dovede vás to zpět tam, kde právě jste. Oceňujete kód, který vám někdo zanechal – kódem, zanechaným někým, kdo byl v tomto oboru velmi pečlivý, a skláníte se před ním.



Jedno slovo: péče. To je skutečné téma této knihy. Možná, že vhodný podtitul měl být *Jak pečovat o kód*.

Michael uhodil hřebík na hlavičku. Čistý kód je takový kód, který jeho autor vytvořil s pečlivostí. Někdo si udělal čas, aby jej udržoval jednoduchý a upravený. Věnoval přiměřenou pozornost detailům. Pečoval o něj.

4. [Knuth92].

## Ron Jeffries, autor knih „Extreme Programming Installed“ a „Extreme Programming Adventures in C#“

Ron začal svou dráhu programátora ve Fortranu ve Strategickém velitelství vzdušných sil a psal kód téměř v každém jazyce a pro téměř každý počítač. Stojí za to pozorně zvažovat jeho slova.

*V současné době začínám a v podstatě i končím s Beckovými pravidly pro jednoduchý kód. V pořadí podle důležitosti, jednoduchý kód:*

- ◆ *Projde všemi testy;*
- ◆ *neobsahuje žádná opakování kódu;*
- ◆ *vyjadřuje veškeré myšlenky návrhu, které jsou v systému;*
- ◆ *minimalizuje počet entit, jako jsou třídy, metody, funkce a podobně.*



*Z nich se nejvíce soustředím na opakování kódu. Pokud se něco neustále opakuje, je to známka, že máme v hlavě myšlenku, která není v kódu dobré vyjádřena. Pokouším se zjistit, co to je. Pak se pokouším tuto myšlenku vyjádřit jasněji.*

*Schopnost vyjádřit se v sobě zahrnuje smysluplná jména a já jsem schopen změnit jména i několikrát, dokud nejsem spokojený. U moderních kódovacích nástrojů, jako je Eclipse, je přejmenování celkem nenáročné, takže mi změny neprinášejí nějaké těžkosti. Schopnost vyjádřit se v sobě nezahrnuje jen jména. Sleduj také, zda objekt nebo metoda provádí více než jednu věc. Je-li to objekt, bude pravděpodobně nutné rozdělit ho na dva nebo více objektů. Je-li to metoda, vždy ji budu refaktorovat pomocí extrakce metody a výsledkem bude jediná metoda, u které bude jasnější, co dělá, a několik podřízených metod vysvětlujících, jak se to dělá.*

*Zdvojování a schopnost vyjádřit se pro mě znamenají dlouhou cestu k tomu, co považuji za čistý kód. Zlepšování nečistého kódu jen témito dvěma prostředky může znamenat veliké zlepšení. Je tu ale ještě jedna věc, které jsem si vědom a již je poněkud těžké vysvětlit.*

*Po letech, během kterých jsem pracoval v tomto oboru, se mi zdá, že všechny programy se skládají z velmi podobných prvků. Například „najdi položky v kolekci“. Ať už máme databázi se záznamy zaměstnanců, hešovou mapu klíčů a hodnot nebo pole nějakých položek, obvykle požadujeme nějakou konkrétní položku z této kolekce. Když k tomu dojde, obvykle zabalím tuto konkrétní implementaci do abstraktnější metody nebo třídy. To mi poskytuje řadu zajímavých výhod.*

*Tuto funkcionalitu mohu nyní implementovat s něčím jednoduchým, třeba s hešovou mapou, ale protože jsou takto veškeré odkazy na prohledávání řešeny pomocí mé drobné abstrakce, mohu změnit implementaci, kdykoliv to bude potřeba. Mohu postupovat rychle vpřed a zároveň zachovat schopnost měnit kód v budoucnu.*

*Navíc, abstrakce kolekce často upoutá mou pozornost na to, co se „skutečně“ děje, a nemusím přitom přistupovat k implementaci libovolné funkčnosti kolekce, když vše, co doopravdy potřebuji, je pár celkem jednoduchých způsobů, jak nalézt to, co potřebuji.*

*Omezení opakování kódu, vysoká vypovídací schopnost a včasné vytváření jednoduchých abstrakcí. To pro mě znamená čistý kód.*

Zde shrnul Ron v několika krátkých odstavcích obsah této knihy. Žádná zdvojení, jedna věc, expresivita, krátké abstrakce. Vše je zde zahrnuto.

## **Ward Cunningham, tvůrce „Wiki“, tvůrce nástroje „Fit“, spolutvůrce „eXtrémního Programování“. Motivační síla v pozadí „Návrhových vzorů“. Čelný myšlenkový představitel objektově orientovaného programování a jazyka Smalltalk. Kmotr všech, kteří se zajímají o kód.**

*Že pracujete s čistým kódem, poznáte podle toho, že každá procedura, kterou procházíte, se ukáže být tím, co jste do značné míry předpokládali. Kód můžete označit za nádherný, když vypadá, jako kdyby byl použitý jazyk pro daný problém stvořen.*

Podobné výroky jsou pro Warda charakteristické. Čtete je, kýváte hlavou a pak se věnujete dalšímu tématu. Zní natolik rozumně, tak samozřejmě, že to stěží považujete jako něco hlubokého. Možná si pomyslíte, že to je něco, co jste docela dobře předpokládali. Ale podívejme se na to podrobněji.



„...co jste do značné míry předpokládali.“ Kdy jste viděli napsely nějaký modul, který by byl takový, jak jste předpokládali?

Není to spíše tak, že moduly, které různě procházíte, jsou záhadné, komplikované, zamotané? Není chybný směr spíše pravidlem? Nejste spíše zvyklí chodit sem a tam a zkoušet zachytit nějaký praměnek dedukce, který vyvěrá z celého systému a proplétá si cestu skrz modul, který studujete? Kdy to bylo naposledy, co jste procítali nějaký kód a kývali jste hlavou tak, jako jste možná kývali při čtení Wardových tvrzení?

Ward předpokládá, že když čtete čistý kód, nic vás nepřekvapí. Opravdu, nebudeste dokonce potřebovat příliš mnoho úsilí. Budete jej číst a bude to do značné míry to, co očekáváte. Bude samozřejmý, jednoduchý a podmanivý. Každý modul bude lešením pro modul následující. Každý vám řekne, jak bude ten další napsán. Programy, které jsou *takto* čisté, jsou napsány tak důkladně a dobře, že si toho ani nevšimnete. Návrhář je dělá tak, že vypadají absurdně jednoduše, jako všechny výjimečné návrhy.

A co říkáte Wardově poznámce o kráse? Všichni kritizujeme skutečnost, že naše jazyky nebyly navrženy pro naše problémy. Ale Wardovo tvrzení vrací břemeno zpět na nás. Tvrdí, že u krásného kódu *náš jazyk vypadá, jakoby byl vytvořen pro náš problém!* Takže je to *naše* povinnost, aby v našich rukách vypadal jazyk jednoduše! Mějte se na pozoru, ať už jste kdekoliv, fanatici jazyka! Není to jazyk, díky kterému vypadá program jednoduše, ale programátor, jenž jej jednoduchým vytváří!

## **Myšlenkový směr**

A co já (strýček Bob)? Co si myslím o čistém kódu já? Tato kniha vám bude vyprávět až do ohyzdých detailů, co si já a moji krajané o čistém kódu myslíme. Sdíleme vám svůj názor na to, co je to čisté jméno proměnné, čistá funkce, čistá třída atd. Představíme vám tyto názory jako absolutní prav-

dy a nebudeme se omlouvat za naši úpornost. V těchto okamžicích našich profesních drah to *jsou* absolutní pravdy. Představují *naše učení* o čistém kódu.

Vyznavači bojových umění se neshodují v tom, které bojové umění je to nejlepší nebo jaká technika v daném bojovém umění je nejlepší. Mistr bojových umění často zakládá svůj vlastní systém a shromažďuje kolem sebe studenty, aby se od něj učili. Takže máme *Gracie Jiu Jitsu*, založené a vyučované rodinou Gracie v Brazílii. Máme *Hakkoryu Jiu Jitsu*, které založil a vyučoval Okuyama Ryuho v Tokiu. Máme *Jeet Kune Do*, které založil a vyučoval Bruce Lee ve Spojených státech.



Studenti těchto směrů jsou pohrouzeni do výuky svých zakladatelů. Věnují se učení toho, co vyučuje jednotlivý mistr, a často jsou učení jiných mistrů vyloučena. Později, když se studenti ve svém umění zlepšíjí, se mohou stát studenty jiného mistra a mohou své znalosti a výcvik rozšířit. Některí si nakonec vypilují své dovednosti, objeví nové techniky a založí své vlastní školy.

Žádná z těchto škol není absolutně *správná*. Ale přesto v rámci konkrétní školy *jednáme* tak, jako kdyby výuka a techniky *byly* správné. Nakonec existuje správný způsob, jak cvičit Hakkoryu Jiu Jitsu nebo Jeet Kune Do. Ale správnost v rámci jedné školy neznehodnocuje učení jiných škol.

Považujte tuto knihu za popis *Objektové výcvikové školy čistého kódů*. Techniky a učení v ní obsažené jsou způsobem, kterým *my* praktikujeme *naše* umění. Jsme ochotni tvrdit, že pokud se budete touto naukou řídit, budete se těsit z jeho přínosu tak, jako jsme se těšili my, a naučíte se psát kód, který je čistý a profesionální. Ale nemyslete si, že jsme jaksi „pravdiví“ v absolutním slova smyslu. Existují jiné školy a jiný mistři, kteří si mohou dělat nároky na profesionalismus tak jako my. Je vhodné se učit i od nich.

Skutečně, mnohá doporučení z této knihy jsou kontroverzní. Nebudete pravděpodobně souhlasit se všemi. Možná, že některá rozhodně odmítnete. To je v pořádku. Netvrdíme, že jsme konečnou autoritou. Na druhé straně jsou doporučení v této knize tím, o čem jsme již dlouho a úmorně uvažovali. Během desetiletí jsme se učili prostřednictvím zkušenosti a opakovanými pokusy a chybami. Takže ať souhlasíte či nesouhlasíte s čímkoliv, bylo by hanbou nepoznat a nerespektovat náš názor.

## Autoři

Pole @author v nástroji Javadoc říká, kdo jsme. Jsme autoři. A k autorům patří, že mají čtenáře. Skutečně, autoři jsou *zodpovědní* za dobrou komunikaci se čtenáři. Až napišete příště rádek kódu, pamatujte na to, že jste autoři píšící pro čtenáře, kteří budou hodnotit vaše úsilí.

Můžete se ptát: Do jaké míry je kód opravdu čten? Nevěnujeme většinu úsilí jeho psaní?

Přehrávali jste si někdy zpětně postup při editaci? V osmdesátých a devadesátých letech jsme měli editory jako Emacs, který si dokázal pamatovat každý jednotlivý úhoz. Mohli jste pracovat třeba hodinu a pak si zpětně přehrát celý postup jako u zrychleného filmu. Když jsme to udělali, výsledek byl fascinující. Drtivá většina přehrávky spočívala v rolování a v pohybu v jiných modulech!

*Bob otevřá modul.*

*Roluje textem směrem dolů, aby našel funkci, kterou potřebuje změnit.*

*Zastavuje se a zvažuje alternativy.*

*Hm, roluje na začátek modulu, aby si ověřil inicializaci proměnné.*

*Nyní opět roluje dolů a začíná psát.*

*Ale, nyní to, co napsal, maže!*

*Píše to znovu.*

*Znovu to maže!*

*Píše polovinu něčeho jiného, ale pak to opět maže!*

*Přemisťuje se dolů na jinou funkci, jež volá funkci, kterou právě edituje, aby se podíval na její jméno.*

*Roluje opět nahoru a vkládá ten samý kód, který právě smazal.*

*Dělá si přestávku.*

*Znovu ten kód maže!*

*Aktivuje jiné okno a divá se na podtržidu. Je tato funkce překrytá?*

Chápete, oč tu běží. Skutečně, poměr času, stráveného čtením k času strávenému psaním je více než 10:1. Neustále čteme starý kód jako součást úsilí při psaní nového kódu.

Protože je tento poměr tak vysoký, chceme, aby bylo čtení kódu jednoduché, i když psaní takového kódu je těžší. Samozřejmě že neexistuje způsob, jak psát kód, aniž bychom jej četli, *takže kód, který se snadněji čte, se také snadněji píše*.

Z této logiky není úniku. Nemůžete psát kód, pokud nejste schopni čist okolní kód. Psaní kódu, který zkoušíte psát dnes, bude složité nebo jednoduché v závislosti na tom, jak obtížně nebo snadno budete čist okolní kód. Pokud tedy chcete postupovat rychle, pokud chcete být hotovi v krátkém čase, pokud chcete psát váš kód jednoduše, pište jej tak, aby se dal snadno číst.

## Skautské pravidlo

Nestačí psát dobře kód. Je nutné jej v průběhu času *udržovat čistý*. Všichni jsme viděli, že kód může s postupujícím časem degenerovat. Takže v prevenci této degradace musíme hrát aktivní roli.

Američtí skauti mají jednoduché pravidlo, které lze v našem oboru použít.

*Zanechte tábořiště čistší, než jste jej nalezli<sup>5</sup>.*

Kdybychom se podívali dovnitř kódu a nechali jej o něco čistší, než když jsme jej odložili, kód by jednoduše nemohl degenerovat. Úklid nemusí být něco velkého. Změna jednoho jména proměnné k lepšímu, rozdelení jedné funkce, která je poněkud velká, odstranění části zdvojení, pročistění jednoho složeného příkazu if.

5. Jde o adaptaci věty na rozloučenou, kterou vzkazuje skautům Robert Stephenson Smyth Baden-Powell: „Zkuste zanechat tento svět o něco lepší, než jste jej nalezli...“

Umíte si představit práci na projektu, kde se kód s časem *jednoduše zlepšuje*? Věříte, že jakákoliv jiná alternativa je profesionální? Skutečně, není neustálé zdokonalování pravou podstatou profesionalismu?

## Úvod a principy

V mnoha ohledech je tato kniha „úvodem“ ke knize, kterou jsem napsal v roce 2002. Má název „*Agile Software Development: Principles, Patterns, and Practices*“ (PPP). Kniha PPP se zabývá principy objektově orientovaného návrhu a mnohými pravidly profesionálních vývojářů. Jestliže jste tuto knihu nečetli, můžete později zjistit, že na vyprávění této knihy navazuje. Pokud jste ji již četli, zjistíte, že mnoho názorů v ní obsažených má svou odezvu na úrovni kódu i zde.

V této knize naleznete sporadické odkazy na různé zásady návrhu. Ty obsahují mimo jiné *princip jedné odpovědnosti* (Single Responsibility Principle – SRP), *princip otevřenosti a uzavřenosti* (Open Closed Principle – OCP) a *princip inverze závislosti* (Dependency Inversion Principle – DIP). Tyto zásady jsou podrobně popsány v knize PPP.

## Závěr

Knihy o umění neslibují, že z vás udělají umělce. Vše, co vám mohou poskytnout, jsou některé nástroje, techniky a myšlenkové procesy, které používají ostatní umělci. Ani tato kniha vám nemůže slíbit, že z vás udělá dobrého programátora. Nemůže vám slíbit, že vám poskytne „smysl pro kód“. Může vám pouze ukázat myšlenkový postup dobrého programátora a finty, techniky a nástroje, které používají.

Tak jako kniha o umění, i tato kniha bude plná detailů. Bude obsahovat spoustu kódu. Uvidíte dobrý kód a také špatný kód. Uvidíte špatný kód, jak se transformuje v dobrý kód. Uvidíte seznamy heuristik, postupů a technik. Uvidíte příklad za příkladem. A pak už je to na vás.

Rád bych připomněl jeden starý vtip o houslistovi, který cestou na své vystoupení zbloudil. Na rohu potká staršího muže a ptá se jej, jak se dostane do Carnegie Hall. Muž se podívá na něj a pak na housle, které má zastrčené pod paží a říká: „Musíš cvičit, synu, cvičit!“

## Použitá literatura

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

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

## **vqbnhq s bovÙ**

*Journal of Water Health* is a refereed journal that covers:

四百五

Consequently, the results of this study indicate that the use of a single species of animal as a primary indicator of environmental quality may not be sufficient to predict the effects of environmental change on the entire ecosystem.

**utské pravidlo**

## **Ústeké pravidlo**

pozitivní doba a když je význam jeho výroby i na většinu svého života.

1905 ခုနှစ်၊ မြန်မာနိုင်ငံ၊ ရန်ကုန်မြို့၊ အမြန် ၁၂၃၀၈။

प्राप्ति विवरण विवरण विवरण विवरण विवरण विवरण विवरण विवरण

# KAPITOLA 2

## Smysluplná jména

*Tim Ottlinger*

### V této kapitole najdete:

- ◆ Úvod
- ◆ Používejte jména vysvětlující význam
- ◆ Vyhnete se dezinformacím
- ◆ Dělejte smysluplné rozdíly
- ◆ Používejte vyslovitelná jména
- ◆ Používejte jména, která lze vyhledat
- ◆ Vyhnete se skrytému překládání jmen
- ◆ Jména tříd
- ◆ Jména metod
- ◆ Nesnažte se být strojení
- ◆ Volte jedno slovo pro jeden pojem
- ◆ Nepoužívejte slovní hříčky
- ◆ Používejte jména z domény řešení
- ◆ Používejte jména domén problému
- ◆ Přidejte smysluplné souvislosti
- ◆ Nepřidávejte kontext bezdůvodně
- ◆ Slovo na závěr



## Úvod

Všude v softwaru narazíme na jména. Pojmenováváme své proměnné, funkce, argumenty, třídy a balíčky. Pojmenováváme zdrojové soubory a také adresáře, ve kterých se nacházejí. Pojmenováváme soubory jar, war a ear. Pojmenováváme a pojmenováváme. Protože to děláme tak často, měli bychom to dělat dobré. Takže si ukážeme několik jednoduchých pravidel pro tvorbu dobrých jmen.

## Používejte jména vysvětlující význam

Je jednoduché říci, že jména mají vysvětlovat význam. Co bychom vám chtěli vštípit je, že to míníme naprostě vážně. Výběr dobrých jmen zabere čas, ale ušetří mnohem více času, než kolik ho při tom strávíte. Vyberejte tedy jména opatrně a změňte je, když přijdete na lepší. Když to budete dělat, bude každý (včetně vás), kdo váš kód přečte, spokojenější.

Jméno proměnné, funkce nebo třídy by mělo zodpovědět všechny zásadní otázky. Mělo by vám říci, proč existuje, co dělá a jak se používá. Pokud vyžaduje jméno poznámku, pak svůj význam neodkrývá.

```
int d; // elapsed time in days
```

Jméno d neodhaluje nic. Nevyvolává pocit uplynulého času nebo dnů. Měli bychom si vybrat jméno, které specifikuje, co měříme, a jednotku tohoto měření.

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Výběr jména, které odhaluje význam, může velmi usnadnit pochopení kódu a provádění jeho změn. Jaký je smysl následujícího kódu?

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

Proč je tak obtížné říci, co tento kód dělá? Nejsou tam žádné obtížné vzorce. Proložení mezer a od-sazování je logické. Jsou zde pouze tři proměnné a dvě konstanty. Nejsou zde dokonce žádné efektní třídy nebo polymorfni metody, jako je třeba seznam polí (nebo to tak alespoň vypadá).

Problém není v jednoduchosti kódu, ale v jeho podstatě, což je (jak se říká): stupeň, do kterého není kontext vyjádřený samotným kódem. Kód mlčky vyžaduje, abychom znali odpovědi na tyto otázky:

1. Jaký druh položek obsahuje theList?
2. Jaký je význam nultého indexu položky v theList?
3. Jaký je význam hodnoty 4?
4. Jak bych mohl využít vraceného seznamu?

V uvedeném příkladu kódu odpovědi na tyto otázky nenalezneme, ale mohly by tam být. Řekněme, že pracujeme na hře Hledání min. Zjistíme, že na panelu je seznam buněk jménem `theList`. Přejměnujme jej na `gameBoard`.

Každá buňka na panelu je reprezentovaná jednoduchým polem. Dále zjišťujeme, že nultý index znamená umístění stavové hodnoty a že stavová hodnota 4 znamená „označeny“. Kód můžeme znatelně vylepšit pouhým zavedením výmluvných jmen:

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

Všimněte si, že jednoduchost kódu se nezměnila. Má stejný počet operátorů a konstant se stejným počtem vnořených úrovní. Ale kód je nyní mnohem jasnější.

Můžeme jít dále a místo používání pole s položkami typu `integer` napsat jednoduchou třídu pro buňky. Může obsahovat funkci (říkejme ji `isFlagged`) s názvem odhalujícím význam, která má skrýt magic-ká čísla. Nová verze funkce vypadá takto:

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

S těmito jednoduchými změnami ve jménech není obtížné porozumět, o co zde jde. To je síla správné volby jmen.

## Vyhňete se dezinformacím

Programátoři si musí dát pozor a nezanechávat falešné stopy, které zatemňují význam kódu. Měli bychom se vyhnout používání slov, jejichž zařízení význam se liší od toho, co chceme vyjádřit. Například jména `hp`, `aix` nebo `sco` jsou špatná, protože se jedná o názvy z unixové platformy nebo jejích variant. I když třeba hledáte v kódu jméno pro přeponu (anglicky hypotenuse) a `hp` vypadá jako dobrá zkratka, mohla by být zavádějící.

Neodkazujte se na skupinu účtů jako na `accountList`, pokud to opravdu není `List`. Tohle slovo znamená pro programátory něco specifického. Pokud není kontejner s uloženými účty opravdu `List`, může to vést k nesprávným závěrům.<sup>1</sup>

Bude lepší použít jména jako `accountGroup` nebo `bunchOfAccounts` nebo pouze `accounts`.

1. Jak uvidíme dále, i když kontejner je `List`, bude pravděpodobně lepší nevkládat jeho typ do jména

Nepoužívejte jména, která se liší jen nepatrně. Jak dlouho vám zabere zjistit drobný rozdíl mezi XYZ-ControllerForEfficientHandlingOfStrings v jednom modulu a XYZControllerForEfficientStorageOfStrings o kousek dálé? Obě slova mají strašně podobný tvar.

Zápis podobných pojmu podobným způsobem je *informace*. Používání nekonzistentního způsobu zápisu je *dezinformace*. V moderních prostředích Javy využíváme automatické dokončování slov. Napišeme několik znaků jména a stiskneme nějakou horkou kombinaci kláves (je-li k dispozici) a za odměnu obdržíme seznam všech možných variant tohoto jména. Je to velmi užitečné, pokud jsou jména podobných pojmu setříděna podle abecedy a pokud jsou rozdíly mezi nimi samozřejmé, protože vývojář má tendenci si vybírat objekt podle jména, aniž by se díval na množství komentářů nebo dokonce na seznam metod obsažených ve třídě.

Opravdu strašný příklad zavádějících jmen proměnných by bylo používání malého písmene „l“ nebo velkého „O“ a zvláště jejich kombinací. Problém je samozřejmě v tom, že se příliš podobají konstantám jedna a nula.

```
int a = 1;
if ( 0 == 1 )
a = 01;
else
1 = 01;
```

Čtenář by si mohl myslit, že je to výmysl, ale prověrovali jsme kód, kde se to takovými jmény jen hemžilo. V jednom případě navrhoval autor kódu, aby se používal jiný font, aby byly rozdíly viditelnější. Což je řešení, které by se muselo předávat všem budoucím vývojářům formou ústní tradice nebo nějakého písemného dokumentu. Problém lze vyřešit s konečnou platností a bez další nové práce pouhým přejmenováním.

## Dělejte smysluplné rozdíly

Programátoři si vytvářejí problémy sami, když píšou kód, jen aby vyhověli požadavkům překladače nebo interpretu. Když například nelze v jednom oddíle kódu použít stejné jméno pro dvě různé položky, můžete mít nutkání změnit jedno z nich podle své nálady. Někdy se to dělá chybným hláskováním jednoho z nich, což ale vede k neočekávaným situacím, kdy oprava téhoto chyb způsobí chyby při překladu.<sup>2</sup>



Pojmenování číselných posloupností (a1, a2, ... aN) je pravým opakem smysluplných jmen. Taková jména nejsou dezinformační, ale nic neříkající. Neposkytují ani náznak autorova úmyslu. Podívejme se na tohle:

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

2. Vezměte například opravdu špatnou praxi pojmenovat proměnnou klass jen proto, že slovo class již bylo použito někde jinde.

Tato funkce se bude číst daleko lépe, když jako jména argumentů použijete source a destination.

Bezvýznamná slova, v podstatě šum, znamenají další bezobsažné rozlišení. Představte si, že máte třídu Product. Máte-li jinou třídu s názvem ProductInfo nebo ProductData, máte dvě různá jména, která ale znamenají totéž. Slova Info a Data jsou v podstatě šum, jako jsou neurčité členy a, an nebo člen určitý the.

Všimněte si, že není nic špatného na používání konvencí pro předložky, jako je a nebo the, pokud znamenají smysluplné rozlišení. Například můžete používat předponu a pro všechny lokální proměnné a the pro všechny argumenty funkcí.<sup>3</sup>

Problém nastává, když se rozhodnete nazývat proměnnou theZork jen proto, že již máte jinou proměnnou s názvem zork.

Slova představující šum jsou redundantní. Slovo variable by se nikdy nemělo objevit jako jméno proměnné. Slovo table by se nikdy nemělo stát součástí jména tabulky (table). Oč je NameString lepší než Name? Může někdy být Name číslo v pohyblivé řádové čárce? Pokud ano, porušuje to dřívější pravidlo ohledně dezinformace. Představte si, že naleznete třídu jménem Customer a druhou se jménem CustomerObject. Proč byste to měli brát jako rozdílné? U kterého z nich najdete nejlépe historii zákazníkových plateb?

Víme o jedné aplikaci, která je toho dobrým příkladem. Změnili jsme jména, abychom skryli viníky, ale zde je přesně ten druh chyby:

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

Jak mají programátoři tohoto projektu vědět, kterou z těchto funkcí mají volat?

Pokud nemáme přesně stanovené konvence, proměnnou moneyAmount nelze odlišit od money, customerInfo nelze odlišit od customer, accountData nelze odlišit od account a theMessage nelze odlišit od message. Rozlišujte jména tak, aby čtenář věděl, k čemu jsou tyto rozdíly dobré.

## Používejte vyslovitelná jména

Lidé jsou dobrí ve zpracování slova. Pro zpracování slov je určena významná část našeho mozku. A slova jsou, jak říká definice, vyslovitelná. Bylo by hanbou nevyužít tak velké části mozku, která se vyvinula proto, aby se zabývala mluveným jazykem. Tvořte tedy jména tak, aby se dala vyslovit.

Pokud je vyslovit nemůžete, nemůžete o nich diskutovat, aniž byste vypadali jako pitomci. „Dobrá, tamhle na bě cé er tří cé en té máme int pé es zet kvé, vidíte?“ Je to důležité, protože programování je společenská činnost.

Jedna společnost, kterou znám, používá jméno genymdhms (generování data, roku, měsíce, hodiny, minuty a sekundy – date, year, month, day, hour, minute, second), takže jdou kolem a říkají „gen ypsilon em dé há em es“. Mám nepříjemný zvyk vyslovovat vše tak, jak je to napsáno, takže jsem začal říkat „genymudahims“. Takto to později nazývala spousta návrhářů a analytiků a stále vypadáme hloupě.

3. Strýček Bob tohle používal v jazyce C++, ale tohoto postupu zanechal, protože v moderních integrovaných vývojových prostředích to není nutné.

Ale měli jsme smysl pro humor a tak to byla legrace. Tak či onak, tolerovali jsme špatná jména. Tyto proměnné bylo nutné novým vývojářům vysvětlit a pak se o tom bavili prostřednictvím hloopě sestavených slov, místo aby používali správné termíny. Srovnejte kód

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
}
```

s kódem

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;;
    private final String recordId = "102";
    /* ... */
}
```

Nyní je možná inteligentní konverzace: „Hej, Miky, podívej se na tento záznam! To generationTimestamp je nastaveno na zítřejší datum! Jak je to možné?“

## Používejte jména, která lze vyhledat

Jména a konstanty, sestávající z jednoho písmene, se v textu obtížně hledají.

Lze jednoduše zadat globální vyhledávání výrazu MAX\_CLASSES\_PER\_STUDENT, ale nalézt číslo 7 bude horší. Prohledávání může vrátit jednotlivé čísla jako součást názvu souboru, jiné definice konstanty a různé výrazy, kde se tato hodnota používá za jiným účelem. Ještě horší případ nastává, když má konstanta dlouhé číslo a dojde k transpozici cifer, čímž vznikne programová chyba a konstantu se nepodaří najít.

Stejně tak je jméno e špatnou volbou pro kteroukoliv proměnnou, již by programátor chtěl hledat. Nejen v angličtině to je nejfrekventovanější písmeno a je pravděpodobné, že se objeví v jakékoli části textu jakéhokoli programu. V tomto ohledu jsou delší jména lepší než kratší a každé jméno, které lze v kódu vyhledávat, je lepší než konstanta.

Osobně dávám přednost tomu, aby jména tvořená jedním písmenem byla použita JEN pro lokální proměnné uvnitř krátkých metod. *Délka jména by měla odpovídat délce použitého kódu [Jm5]*. Pokud má být proměnná nebo konstanta vidět, nebo se má používat na různých místech kódu, je nezbytné jí dát jméno, které se bude snadno vyhledávat. Porovnejme znovu

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

s kódem

```
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;  
}
```

Všimněte si, že `sum` není nijak zvlášť vhodné jméno, lze ho ale aspoň hledat. Kód, používající jména s nějakým zámčrem, bude délku funkce prodlužovat, ale zvažte, oč jednodušší bude hledat `WORK_DAYS_PER_WEEK` než prohledávat všechna místa, kde byla použita číslice pět, a pak vyfiltrovat z jejich seznamu jen na ta, která mají zamýšlený význam.

## Vyhñe se kódování jmen

Máme dost práce s kódováním, takže není třeba přidělávat si ještě další práci. Vkládat do jmen informaci o typu nebo rozsahu prostě jen přidává další břemeno při jejich dekódování. Stěží bude rozumné učit každého nového zaměstnance novému kódovacímu „jazyku“, navíc když se musí naučit (obvykle značnému) rozsahu kódu, se kterým budou pracovat. Ve snaze vyřešit problém je to zbytečné duševní břemeno. Zakódovaná jména se obtížně vyslovují a je snadné si splést jejich znění.

### Maďarská notace

Z starých časů, když jsme pracovali s jazyky s omezenou délkou jmen, jsme toto pravidlo porušovali z nutnosti a s politováním. Fortran nás nutil kódovat jména tím, že první písmeno bylo příznakem typu. První verze jazyka BASIC umožňovaly použít jako jméno jen písmeno plus jednu číslici. Maďarská notace (HN – Hungarian Notation) to převedla na novou úroveň.

Maďarská notace byla v časech používání Windows API v jazyce C pokládána za poměrně důležitou. V jazyce C bylo vše buď identifikační číslo typu integer (integer handle), nebo vzdálený ukazatel, ukazatel typu `void*` nebo některý z „řetězců“ (s různým použitím a atributy). Tehdy překladač nekontroloval typy, takže programátoři potřebovali berličku, která by jím pomohla zapamatovat si typy.

V moderních jazycích máme mnohem lepší typové systémy, překladače si je pamatuji a vnucují nám je. Navíc, trendem je psát menší třídy a kratší funkce, takže ve většině případů mohou lidé deklaraci každé použité proměnné vidět.

Programátoři v jazyce Java nepotřebují typ kódovat. Java je silně typový jazyk a editovací prostředí pokročila natolik, že umějí detekovat chybu v typu mnohem dříve, než spustíte překlad! Dnes je maďarská notace a jiné formy kódování typu jednoduše na závadu. Změna jména nebo typu proměnné, funkce nebo třídy je u nich obtížnější. Kód je hůře čitelný. A způsobuje, že kódovací systém může čtenáře zmást.

```
PhoneNumber phoneString;  
// Jméno se při změně typu nemění!
```

### Členské předpony

Také předponu `m_` ke jménům členských proměnných již nemusíte přidávat. Vaše třídy a funkce by měly být dostatečně krátké, abyste je nepotřebovali. A měli byste používat editovací prostředí, které zvýrazňuje nebo podbarvuje členské proměnné, aby se odlišovaly.

```

public class Part {
    private String m_desc; // Textový popis
    void setName(String name) {
        m_desc = name;
    }
}

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

```

Kromě toho se lidé rychle naučí předponu (nebo příponu) ignorovat a vidět jen smysluplnou část jména. Čím více kód čteme, tím méně sledujeme předpony. Nakonec se předpony stanou neviditelnou změtí a příznakem zastaralého kódu.

## Rozhraní a implementace

Někdy se používají zvláštní případy kódování. Řekněme například, že vytváříte abstraktní továrnu (ABSTRACT FACTORY) pro tvorbu tvarů. Tento objekt bude představovat rozhraní a bude implementován v konkrétní třídě. Jak byste je měli pojmenovat? IShapeFactory, nebo ShapeFactory? Raději nechávám jméno rozhraní neozdobené. Počáteční I, tak obvyklé v dnešních balících zděděného kódu, je v nejlepším případě rušivé a v tom nejhorším znamená příliš mnoho informace. Nechci, aby moji uživatelé věděli, že jim odevzdávám rozhraní. Pouze chci, aby věděli, že je to ShapeFactory. Takže pokud musím zakódovat rozhraní nebo implementaci, vyberu si implementaci. Název ShapeFactoryImp nebo dokonce nehezké CShapeFactory je lepší, než kódovat rozhraní.

## Vyhñe se skrytému překládání jmen

Čtenáři by si neměli v duchu překládat vaše jména na jiná, která již znají. Tento problém obvykle nastává, když nechceme použít ani termíny domény oblasti problému, ani termíny z oblasti řešení.

Je to problém jmen proměnných, které se skládají z jediného písmene. Čítač cyklu lze určitě nazvat písmeny i nebo j nebo k (ale nikdy 1!), je-li rozsah opravdu malý a žádná jiná jména s ním nebudou v konfliktu. Je to proto, že jména s jedním písmenem jsou zde tradici. Ale ve většině jiných souvislostí není název s jedním písmenem dobrou volbou. Je to pouze zástupný název, který si musí čtenář v duchu přiřadit k aktuálnímu pojmu. Používám-li jako jméno písmeno c jen proto, že se písmena a nebo b již používají, pak je to ten nejhorší důvod, jaký může být.

Obecně jsou programátoři celkem bystří lidé. Bystří lidé někdy rádi svou inteligenci ukazují tím, že predvádějí své schopnosti žonglování. Koneckonců, pokud jste schopni si spolehlivě zapamatovat, že r reprezentuje verzi url, psanou malými písmeny bez uvedení schématu a hostujícího počítače, pak jste zřejmě velmi bystří.

Jeden rozdíl mezi bystrým a profesionálním programátorem je, že profesionál chápe, že jasnost je nad vše. Profesionálové používají svých schopností permanentně a píšou kód, kterému ostatní mohou rozumět.

## Jména tříd

Třídy a objekty mohou obsahovat podstatná jména nebo spojení podstatných jmen, jako Customer, WikiPage, Account nebo AddressParser. V názvech tříd se vyhněte slovům, jako je Manager, Processor, Data nebo Info. Jméno třídy by nemělo být sloveso.

## Jména metod

Metody by měly obsahovat slovesa nebo slovesné fráze, jako třeba postPayment, deletePage nebo save. Přístupové objekty, mutátory a predikáty by měly být pojmenovány podle své hodnoty a s předponou get, set a is podle standardu pro komponenty javabeans.<sup>4</sup>

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

U přetěžovaných konstruktorů použijte statické tovární metody se jmény, které popisují argumenty. Například

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

je obecně lepší, než

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

Zvažte, zda by nebylo vhodné prosadit jejich používání tím, že odpovídající konstruktory budou privátní.

## Nesnažte se být strojení

Jsou-li jména příliš důmyslná, budou zapamatovatelná jen pro ty, kteří sdilejí váš smysl pro humor, a to jen tak dlouho, dokud si budou pamatovat váš vtip. Budou vědět, co dělá funkce HolyHandGrenade („svatý ruční granát“)? Jistě to je šikovné, ale v tomto případě by mohlo být lepší jméno DeleteItems („odstraň položky“). Dávejte přednost jasnosti před pobavením. Strojenost se v kódu často projevuje ve formě hovorové řeči nebo slangu. Nepoužívejte například jméno whack() (useknout) tam, kde má být kill(). Nepoužívejte vtipná spojení, závislé na kultuře, jako je eatMyShorts() (sněz mi moje kraťasy), což znamená abort().

Říkejte to, co máte na mysli a myslíte vážně to, co říkáte.



## Volte jedno slovo pro jeden pojem

Vyberte si jedno slovo pro každý pojem a zůstaňte u něj. Například je matoucí mít slova fetch, retrieve a get jako ekvivalentní metody v různých třídách. Jak si budete pamatovat, které jméno metody

4. <http://java.sun.com/products/javabeans/docs/spec.html>.

patří do které metody? Bohužel si často budete muset vzpomínat, která společnost, skupina nebo jaký jednotlivec napsal knihovnu či třídu, abyste si zapamatovali, který termín byl použit. Jinak strávíte obrovské množství času prohledáváním hlavičkových souborů a předchozích vzorků kódu.

Moderní editovací prostředí, jako je Eclipse nebo IntelliJ – poskytují kontextově senzitivní vodítka, jako je například seznam metod, které můžete v daném objektu volat. Ale všimněte si, že seznam vám obvykle neposkytne poznámky, které jste si napsali u jmen funkcí nebo seznamů parametrů. Budete mít štěstí, pokud z deklarací funkci dostanete *jména* parametrů. Jméno funkce musí být samostatné a musí být konzistentní v tom smyslu, že obdržíte správnou metodu bez dalšího dodatečného zkoumání.

Podobně je matoucí mít controller, manager nebo driver ve stejném kódu aplikace. Jaký je základní rozdíl mezi DeviceManager a ProtocolController? Proč nejsou názvy obou controller nebo manager? Jsou to opravdu ovladače? Jméno vede k předpokladu, že tyto objekty jsou rozdílné a patří do různých tříd.

Konzistentní slovník je velké dobrodiní pro ty programátory, kteří musí váš kód používat.

## Nepoužívejte slovní hříčky

Vyhnete se používání stejného slova pro dva různé účely. Použití stejného termínu pro dva různé pojmy je v podstatě slovní hříčka.

Budete-li dodržovat pravidlo „jedno slovo pro jeden pojem“, můžete dospět k mnoha třídám, které mají například metodu add. Pokud jsou návratové hodnoty a seznam parametrů u všech metod add sémanticky ekvivalentní, vše je v pořádku.

Někdo by se však mohl rozhodnout používat slovo add z důvodu „konzistence“, i když ve skutečnosti nejde o přidávání ve stejném smyslu. Řekněme, že máme mnoho tříd, kde metoda add vytvoří novou hodnotu sečtením nebo zřetězením dvou existujících hodnot. Řekněme nyní, že píšeme novou třídu s metodou, která ukládá svůj jediný parametr do kolekce. Měli bychom tuto metodu pojmenovat jako add? Mohlo by se to zdát konzistentní, protože máme tolik jiných metod add, ale v tomto případě je sémantika jiná, takže bychom ji měli pojmenovat jako insert nebo append. Nazvat tuto metodu jménem add by byla slovní hříčka.

Naším cílem jako autorů je vytvořit natolik srozumitelný kód, nakolik je to jen možné. Chceme, aby čtení našeho kódu bylo rychlým sbíráním smetany a ne předmětem intenzivního studia. Chceme používat model populárního brožovaného vydání, ve kterém autor zodpovídá za to, že je mu rozumět, a ne akademický model, v němž musí žák význam z papírových stránek vydolovat.

## Používejte jména z domény řešení

Nezapomínejte na to, že lidé, kteří budou číst váš kód, budou programátoři. Takže se neostýchejte používat termíny z oboru počítačů, názvy algoritmů, vzorů, matematické termíny atd. Není rozumné vytahovat všechna jména z domény problému, protože nechceme, aby naši spolupracovníci museli běhat sem a tam k zákazníkovi a ptát se ho, co jednotlivá jména znamenají, když tento pojem znají pod jiným jménem.

Jméno AccountVisitor znamená mnoho pro programátora, který je obeznámen s návrhovým vzorem VISITOR. Který programátor by nevěděl, co to znamená JobQueue? Programátoři musí dělat mnoho technických věcí. Volba technického jména pro tyto záležitosti je nejvhodnější.

## Používejte jména domén problému

Jestliže se jméno pro to, co právě děláte, v „programátorštině“ nevyskytuje, použijte název z problémové domény. Alespoň se programátor, který bude udržovat váš kód, může zeptat doménového experta, co to znamená.

Oddělování pojmu z domény řešení a z domény problému je část práce dobrého programátora a návrháře. Kód, který má více co dělat s pojmy problémové domény, by měl mít jména odvozená z ní.

## Přidejte smysluplné souvislosti

Existuje několik jmen, která jsou smysluplná sama od sebe – ale většina z nich není. Naopak, pro vaše čtenáře je potřebujete umístit do kontextu a uzavřít do dobré pojmenovaných tříd, funkcí nebo jmenných prostorů. Když vše selže, jako poslední řešení můžeme použít před jménem předponu.

Představte si, že máte proměnné s názvem `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` a `zipcode`. Když jsou pohromadě, je naprostě jasné, že tvoří adresu. Ale co když uvidíte proměnnou `state` samostatně použitou v metodě? Odvodili byste si automaticky, že se jedná o část adresy? Názvy můžete uvést do souvislosti použitím předpon: `addrFirstName`, `addrLastName`, `addrState` atd. Čtenáři alespoň porozumí tomu, že tyto proměnné jsou součástí větší struktury. Lepší řešením by samozřejmě bylo vytvořit třídu jménem `Address`. Pak i překladač ví, že proměnná patří do širších souvislostí.

Podívejte se na metodu ve výpisu 2.1. Potřebují proměnné smysluplnější kontext? Jméno funkce poskytuje jen část kontextu, algoritmus pak dává zbytek. Když si jednou tyto funkce projdete, uvidíte, že tři proměnné `number`, `verb` a `pluralModifier` jsou součástí zprávy „odhadni statistiku“. Zde je bohužel nutné kontext dedukovat. Při prvním pohledu na metodu jsou významy proměnných nejasné.

### Výpis 2.1. Proměnné s nejasným kontextem

```
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);
```

Funkce je poněkud dlouhá a proměnné jsou používány v celém rozsahu. Abychom funkci rozdělili na menší části, musíme vytvořit třídu `GuessStatisticsMessage` a v ní zřídit tři pole proměnných. Pro tyto tři proměnné to poskytne srozumitelný kontext. Jsou *definitivně* částí třídy `GuessStatisticsMessage`. Zdokonalení kontextu rovněž umožňuje, aby se algoritmus stal mnohem čistějším tím, že bude rozdělen na více malých funkcí. (Viz výpis 2.2.)

#### Výpis 2.2. Proměnné mají kontext

```
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";  
    }  
}
```

## Nepřidávejte kontext bezdůvodně

V imaginární aplikaci jménem „Gas station deluxe“ není dobré dávat každé třídě předponu GSD. Upřímně řečeno byste pracovali v rozporu s posláním vašich nástrojů. Zadáte G, stisknete klávesu pro dokončení vstupu a budete odměněni kilometrem dlouhým seznamem všech tříd systému. Je to rozumné? Proč vývojovému systému jeho pomoc ztěžovat?

Dejme tomu, že jste podobně vymysleli v účtovacím modulu GSD třídu `MailingAddress` a pojmenovali jste ji `GSDAccountAddress`. Později budete potřebovat elektronickou adresu pro aplikaci zpracovávající kontakty se zákazníky. Použijete jméno `GSDAccountAddress`? Zní to jako to správné jméno? Deset ze sedmnácti znaků je nadbytečných nebo nevýznamných.

Kratší jména jsou obecně lepší než delší, pokud jsou srozumitelná. Nepřidávejte ke jménu žádný další kontext, není-li to nezbytné.

Jména `accountAddress` a `customerAddress` jsou dobrá pro instance třídy `Address`, ale mohla by být špatná pro třídy. Jméno `Address` je dobré pro třídu. Pokud potřebuji rozlišit adresu MAC, adresu portu nebo adresu webu, mohl bych uvažovat o `PostalAddress`, `MAC` nebo `URI`. Výsledná jména jsou preciznější, což je hlavním tématem vytváření jmen.

## Slovo na závěr

Na výběru dobrých jmen je nejobtížnější, že vyžaduje dobré popisné schopnosti a sdílení kulturního zázemí. Je to spíše věcí učení než technickou, obchodní nebo řídicí záležitostí. Výsledkem je, že se to v tomto oboru mnoho lidí dělat opravdu dobře nenaučí.

Lidé se také bojí věci přejmenovávat, protože se bojí, že by jiní vývojáři mohli oponovat. My tyto obavy nesdílíme a zjištujeme, že jsme vlastně vděčni, když se jména mění (k lepšímu). Většinu času skutečně jména tříd nebo metod nememorujeme. Používáme moderní nástroje, abychom se vypořádali s detaily a mohli se soustředit na to, zda je možné čist kód jako odstavce a věty nebo alespoň jako tabulky a datové struktury (věta nemusí zobrazovat data tím nejlepším způsobem). Možná, že přejmenováním někoho překvapíte, stejně jako když kód jakkoli jinak vylepsíte. Nenechte se na své dráze zastavit.

Říďte se některými z těchto pravidel a podívejte se, zda tím čitelnost vašeho kódu nezlepšíte. Pokud udržujete kód někoho jiného, použijte k řešení těchto problémů nástroje pro refaktorování. Vyplatí se to krátkodobě a bude se to vyplácet i dlouhodobě.



# KAPITOLA 3

## Funkce

### V této kapitole najdete:

- ◆ Malá!
- ◆ Dělejte jen jednu věc
- ◆ Jedna úroveň abstrakce na funkci
- ◆ Čtení kódu odshora dolů: metoda sestupu
- ◆ Příkazy Switch
- ◆ Používejte popisná jména
- ◆ Argumenty funkcí
- ◆ Žádné vedlejší efekty
- ◆ Oddělování příkazů a dotazů
- ◆ Dejte přednost výjimkám před vracením chybouvých kódů
- ◆ Neopakujte se
- ◆ Strukturované programování
- ◆ Jak napíšete funkci, jako je tato?
- ◆ Závěr
- ◆ Literatura



V raných dobách programování jsme své systémy budovali z rutin a subrutin. Později, v dobách Fortranu a PL/1, jsme je budovali z programů, podprogramů a funkcí. Z těchto počátečních dob přežily do dnešních dnů jen funkce. Funkce jsou velmi důležitou částí organizace jakéhokoli programu. Tématem této kapitoly je, jak je psát dobré.

Podívejte se na výpis 3.1. V nástroji FitNesse<sup>1</sup> je těžké najít nějakou dlouhou funkci, ale po nějakém hledání jsem našel tuhle. Nejen že je dlouhá, ale obsahuje zdvojený kód, mnoho podivných řetězců, mnoho zvláštních a neobvyklých datových typů nebo prvků API. Podívejte se na to, co z toho budete schopni pochopit v následujících třech minutách.

#### Výpis 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 =
```

1. Nástroj s otevřeným zdrojovým kódem. [www.fitnesse.org](http://www.fitnesse.org).

```
    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) {
        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();
}
```

Rozumíte této funkci po třech minutách studia? Pravděpodobně ne. Děje se tam toho příliš mnoho a je tam příliš mnoho různých úrovní abstrakce. Jsou tam podivné řetězce a řada divných volání funkcí promíchaných s vnořenými příkazy i f, řízených příznaky.

Avšak po několika jednoduchých extrakcích metod, několikerém přejmenování a s trohou restrukturování jsem dokázal vystihnout účel této funkce na devíti řádcích výpisu 3.2. Podívejte se, zda dokážete během příštích tří minut porozumět *tomuhle*.

#### Výpis 3.2. HrmlUtil.java (po refaktorování)

```
public static String renderPageWithSetupsAndTear downs(
    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();
}
```

Pokud se nezabýváte FitNesse, pravděpodobně nebudeste rozumět všem detailům. Přesto zřejmě pochopíte, že tato funkce zahrnuje nějaké nastavení a rozdělení stránky do testovací stránky a poté ji vrátí ve formátu HTML. Pokud jste obeznámeni s testovací šablonou JUnit,<sup>2</sup> pravděpodobně jste pochopili, že tato funkce patří do nějaké testovací šablony založené na webové technologii. A to je samozřejmě správně. Je celkem jednoduché tuto informaci vytušit z výpisu 3.2, ale výpis 3.1 ji dost zatemňuje.

Takže proč je funkce z výpisu 3.2 snadno čitelná a srozumitelná? Jak to uděláme, aby funkce dokázala sdělit svůj záměr? Jaké vlastnosti musí naše funkce mít, aby umožnila příležitostnému čtenáři intuitivně pochopit, jaký druh programu v sobě skrývají?

## Malá!

Prvním pravidlem pro psaní funkcí je, že by měly být malé. Druhým pravidlem je, že by měly být ještě menší. Tohle není tvrzení, které bych mohl nějak odůvodnit. Nemohu poskytnout žádné odkazy na výzkum, který dokázal, že velmi malé funkce jsou lepší. Mohu vám říci, že jsem během minulých čtyřiceti let psal funkce všech možných délek. Napsal jsem několik příšerných ohavností, dlouhých na tři tisíce řádků. Napsal jsem funkce v rozsahu sto až tři sta řádků. A také jsem napsal funkce s dvaceti až třiceti řádky. Tato zkušenosť mě naučila metodou pokusů a chyb, že by funkce měly být velmi malé.

V osmdesátých letech jsme říkávali, že by funkce neměla být delší, než je zaplněná obrazovka. To jsme samozřejmě tvrdili v časech, kdy obrazovky VT100 měly 24 řádek a 80 sloupců a naše editory používaly 4 řádky pro administrativní účely. Dnes se zmenšeným fontem a pěkným velkým monitorem můžete umístit 150 znaků na řádku a 100 a více řádek na obrazovku. Řádky by neměly být delší než 150 znaků. Funkce by neměly být dlouhé 100 řádků. Funkce by zřídka měly být dlouhé 20 řádků.

Jak krátká by funkce měla být? Navštívil jsem v roce 1999 Kenta Becka v jeho domě v Oregonu. Sedli jsme si a něco jsme programovali. V jednom okamžiku mi ukázal pěkný malý prográmek v jazyce Java založený na knihovně Swing, který se jmenoval *Sparkle*. Vytvářel na obrazovce vizuální efekt podobný kouzelné hůlce pohádkové kmotry z filmu Cindarella. Když jste pohybovali myší, odletovaly od kurzoru jiskry a s pěkným jiskřením padaly na spodní část okna jakoby skrz gravitační pole. Když mi Kent ukázal kód, byl jsem ohromen, jak byly všechny funkce krátké. Byl jsem zvyklý na funkce z programů používajících Swing, které zabraly kilometry vertikálního prostoru. V tomto programu byla každá funkce dlouhá dva, tři nebo čtyři řádky. Každá z nich byla jasná a samozřejmá. Každá mluvila za sebe. A každá vás naváděla na další funkce podle daného pořadí. *Takhle krátce* by měly vaše funkce být!<sup>3</sup>

Jak krátké by měly být vaše funkce? Měly by být obvykle kratší, než na výpisu 3.2! Opravdu, výpis 3.2 by měl být zkrácen na délku výpisu 3.3.

2. Testovací nástroj pro jednotkové testování Javy s otevřeným zdrojovým kódem, [www.junit.org](http://www.junit.org).

3. Požádal jsem Kenta, zda ještě nemá kopii, ale nemohl žádnou najít. Prohledal jsem své staré počítače, ale bezvýsledně. Z tohoto programu zůstalo vše jen v mé paměti.

**Výpis 3.3. HtmlUtil.java (po refaktorování)**

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

**Bloky a odsazování**

Tohle znamená, že bloky v příkazech `if`, `else`, `while` a jiných by měly mít délku jen jednoho řádku, který by měl pravděpodobně volat funkci. Nejenže funkce v závorkách zůstává krátká, ale tento způsob má rovněž i svou dokumentační hodnotu, protože funkce volaná z bloku může mít dobré popisné jméno.

Také z toho vyplývá, že funkce by neměly být tak velké, aby obsahovaly vnořené struktury. Úroveň odsazení funkce by proto neměla být větší než jedna nebo dvě. To dělá samozřejmě funkci čitelnější a srozumitelnější.

**Dělejte jen jednu věc**

Mělo by být zcela jasné, že kód ve výpisu 3.1 dělá mnohem více než jen jednu věc. Vytváří mimo jiné vyrovnávací paměti, poskytuje stránky, hledá zděděné stránky, poskytuje cesty, přidává tajuplné řetězce a generuje kód HTML. Výpis 3.1 je výpisem kódu, který má spoustu práce, jenž provádí řadu různých úkolů. Obsahuje nastavení a dělení na testovací stránky.

Následující rada se objevuje v různých formách již třicet let, ne-li více.

**FUNKCE BY MĚLA DĚLAT JEN JEDNU VĚC. MĚLA BY JI DĚLAT DOBŘE. NEMĚLA BY DĚLAT NIC JINÉHO.**

Problém s tímto tvrzením je, že je těžké říci, co znamená „jedna věc“. Dělá kód na výpisu 3.3 jednu věc? Je jednoduché přijít na to, že provádí tři věci:

1. Určuje, zda jde o testovací stránku.
2. Pokud ano, zahrne nastavení a dělení.
3. Vrátí stránku v kódu HTML.



Takže, o který případ jde? Dělá tato funkce jednu nebo tři věci? Všimněte si, že tři kroky této funkce jsou na stejně úrovni abstrakce pod uvedeným názvem funkce. Funkci můžeme v angličtině popsat jako krátký paragraf začínající TO:<sup>4</sup>

*TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.*

4. Jazyk LOGO používal klíčové slovo „TO“ stejným způsobem, jako používají jazyky Ruby a Python klíčové slovo „def“. Každá funkce tedy začínala slovem „TO“. Mělo to zajímavý vliv na způsob, jakým byly funkce navrhovány.

Přeložíme-li i jméno funkce, můžeme předchozí odstavec přepsat takto:

*Abychom VykresliliStránkuSNastavenímARozdělením, zkontrolujeme, zda se nejedná o testovací stránku, a pokud ano, zahrneme do ní nastavení a dělení. V každém případě vygenerujeme stránku v kódu HTML.*

Pokud funkce provádí jen ty kroky, které jsou jednu úroveň pod uvedeným názvem funkce, pak tato funkce provádí jednu věc. Nakonec důvodem, proč píšeme funkce, je rozložení rozsáhlého pojmu (jinými slovy názvu funkce) na množinu kroků následující abstrakční úrovňě.

Mělo by být naprostě jasné, že výpis 3.1 obsahuje kroky mnoha různých úrovni abstrakce. To znamená, že zcela jasně dělá více než jednu věc. Výpis 3.2 má dokonce dvě úrovne abstrakce, což lze dokázat tím, že jsme schopni její rozsah snížit. Ale výpis 3.3 by bylo velmi obtížné nějak smysluplně zkrátit. Mohli bychom extrahat příkaz `if` do funkce s názvem `includeSetupsAndTeardownsIfTestPage`, ale to jen zopakuje daný kód, aniž by se úroveň abstrakce změnila.

Jiným způsobem, jak zjistit, že funkce dělá více než „jednu věc“, je extrahat z ní jinou funkci s názvem, který není jen zopakováním předchozí implementace [O34].

## Sekce uvnitř funkcí

Podívejte se na výpis 4.7 na straně 88. Všimněte si, že funkce `generatePrimes` (generování prvočísel) je rozdělena na sekce označené *deklarace*,  *inicializace* a *síto*. Tohle je evidentní symptom, že funkce provádí více než jednu činnost. Funkce, které provádějí jednu věc, nemohou být logicky rozděleny na sekce.

## Jedna úroveň abstrakce na funkci

Abychom měli jistotu, že naše funkce provádějí „jednu věc“, potřebujeme mít jistotu, že příkazy v rámci naší funkce jsou na stejně úrovni abstrakce. Není obtížné zjistit, že výpis 3.1 tohle pravidlo porušuje. Jsou zde koncepty s vysokým stupněm abstrakce, jako třeba `getHtml()`, jiné jsou na střední úrovni, jako třeba `String pagePathName = PathParser.render(pagePath)`, a další, které jsou na nízké úrovni, jako třeba `.append("\n")`.

Míchání úrovní abstrakce v rámci jedné funkce je vždy matoucí. Čtenáři nemusejí být schopni říci, zda konkrétní výraz je základním konceptem nebo detailem. Horší případ nastává, když jsou jednou detaily pomíchány se základními pojmy, více a více detailů vrostete do kódu funkce, což se podobá rozbítým oknům.

## Čtení kódu odshora dolů: metoda sestupu

Kód chceme číst jako vyprávění odshora dolů<sup>5</sup>. Chceme, aby za každou funkci následovaly funkce abstrakce další úrovňě, abychom program mohli číst a během sestupného čtení seznamu funkcí jít v jednom čase o jednu úroveň abstrakce níže. Tohle nazývám *metoda sestupu*.

Jinými slovy, chceme číst kód programu, jako by to byla sada odstavců *TO*, z nichž každý popisuje současnou úroveň abstrakce a odkazuje se na další odstavce *TO* o jednu úroveň níže.

5. [KP78], p. 37.

*Abychom zahrnuli nastavení a dělení, zahrneme nejdříve nastavení, poté zahrneme obsah testovací stránky a poté dělení.*

*Abychom zahrnuli nastavení, zahrneme nastavení sady, jestliže se jedná o sadu, poté zahrneme obvyklé nastavení.*

*Abychom zahrnuli nastavení sady, prohledáme nadřazenou hierarchii stránky „SuiteSetUp“ a přidáme příkaz include s cestou k této stránce.*

*Abychom prohledali nadřazenou...*

Ukazuje se, že pro programátory je velmi obtížné se tohle pravidlo naučit a psát funkce tak, aby zůstaly v jedné úrovni abstrakce. Ale zvládnout tento trik je opravdu velmi důležité. Je to klíčem k tomu, aby funkce zůstaly krátké, a k zabezpečení, že provádějí jen „jednu věc“. Psaní kódu, který lze číst shora dolů jako množinu odstavců TO („ABYCHOM“), je efektivní technikou, jež udržuje úrovně abstrakce konzistentní.

Podívejte se na výpis 3.7 na konci této kapitoly. Ukazuje celou funkci `testableHtml` po refaktorování na základě vše popsaných pravidel. Všimněte si, jak každá funkce uvádí následující a jak každá z nich zůstává na konzistentní úrovni abstrakce.

## Příkazy Switch

Je obtížné napsat krátký příkaz `switch`.<sup>6</sup> I kdyby měl příkaz `switch` jen dva řádky s `case`, je to více, než bych u jednoduchého bloku nebo funkce chtěl vidět. Je také obtížné napsat příkaz `switch`, který by prováděl jen jednu věc. Příkazy `switch` už ze své podstaty provádějí vždy  $N$  úkolů. Bohužel se jim nemůžeme vždy vyhnout, ale můžeme zajistit, že každý příkaz `switch` bude skryt v třídách nízké úrovně a nebude se nikdy opakovat. Je samozřejmé, že k tomu využijeme polymorfismus.

Podívejte se na výpis 3.4. Ukazuje pouze jednu z operací, která může záviset na typu zaměstnance.

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

Tato funkce v sobě skrývá několik problémů. Za prvé je dlouhá a při zvyšování počtu typů zaměstnanců se bude dále zvětšovat. Za druhé se zjevně věnuje více než jedné věci. Za třetí porušuje princip

6. Tím samozřejmě myslím i zřetězené příkazy `if/else`.

jedné odpovědnosti<sup>7</sup> (Single Responsibility Principle – SRP), protože existuje více než jeden důvod ke změnám. Za čtvrté, porušuje princip otevřenosti a uzavřenosti (OCP – Open Closed Principle)<sup>8</sup>, protože kód musí být upravován, kdykoliv přidáme nové typy. Ale možná že nejhorší problém této funkce spočívá v tom, že existuje neomezené množství jiných funkcí, které mohou mít stejnou strukturu. Například bychom mohli mít

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

nebo

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

nebo spoustu jiných. Všechny mohou mít tutéž zhoubnou strukturu. Řešení tohoto problému spočívá v pohřbení tohoto příkazu do suterénu abstraktní továrny (ABSTRACT FACTORY)<sup>9</sup>, aby nebyl nikomu na očích. Abstraktní továrna bude využívat příkaz switch při vytváření příslušných objektů typu Employee, nebo typů odvozených, a různé funkce, jako jsou calculatePay, isPayday, budou polymorfně volány přes rozhraní třídy Employee.

Moje obecné pravidlo pro příkazy switch je, že je lze tolerovat, pokud se objeví jen jednou, používají se pro vytváření polymorfních objektů a jsou skryty za vazbu danou děděním tak, že jsou pro zbytek systému neviditelné [O23]. Samozřejmě že každá situace je jedinečná a existují okamžiky, kdy já sám jednu nebo několik částí tohoto pravidla porušuji.

#### Výpis 3.5. Employee a Factory

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

7. a. [http://en.wikipedia.org/wiki/Single\\_responsibility\\_principle](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](http://en.wikipedia.org/wiki/Open/closed_principle)  
b. <http://www.objectmentor.com/resources/articles/ocp.pdf>.
9. [GOF].

```
        throw new InvalidEmployeeType(r.type);
    }
}
```

## Používejte popisná jména

Ve výpisu 3.7 jsem změnil název funkce z příkladu `testableHtml` na `SetupTeardownIncluder.render`. To je mnohem lepší název, protože lépe popisuje činnost funkce. Také všem soukromým metodám jsem dal popisná jména, jako například `isTestable` nebo `includeSetupAndTeardownPages`. Hodnota dobrých jmen je neocenitelná. Vzpomeňte si na Wardův princip: „*Čistý kód poznáte podle toho, že každá procedura, kterou procházíte, se ukáže být tím, co jste do značné míry předpokládali.*“ Polovina úsilí k dosažení tohoto principu spočívá ve výběru dobrých jmen pro krátké funkce, které dělají jen jednu věc. Čím je funkce kratší a jednoznačněji zaměřená, tím jednodušší bude vybrat pro ni popisné jméno.

Nebojte se vytvářet dlouhá jména. Dlouhé popisné jméno je lepší než jméno krátké a tajuplné. Dlouhé popisné jméno je lepší než dlouhý popisný komentář. Pro vytváření jmen používejte pravidla, která umožňují, aby bylo možné snadno číst složená slova v názvech funkcí a tyto složeniny pak používejte ve jménech, aby tím bylo řečeno, co funkce dělá.

Nebojte se věnovat výběru jména dost času. Měli byste rozhodně vyzkoušet několik různých jmen a s každým z nich si kód přečíst. U moderních integrovaných vývojových prostředí, jako je Eclipse nebo IntelliJ, je změna jména jednoduchá. Použijte některé z těchto prostředí a zkoušejte různá jména, dokud nenaleznete to správné, jež je popisné tak, jak jen to je možné.

Výběr popisného jména projasní návrh modulu ve vaší hlavě a pomůže vám vylepšit ho. Není neobvyklé, že hledání dobrého jména vyústí ve vhodnou restrukturalizaci kódu.

V otázce jmen budete konzistentní. Pro jména funkcí používejte stejně fráze, podstatná jména nebo slovesa, která si vybíráte pro své moduly. Podívejte se například na jména `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` a `includeSetupPage`. Podobná frazeologie jmen umožňuje, aby tato posloupnost dokázala něco sdělit. Kdybych vám opravdu ukázal pouze výše uvedenou posloupnost, ptali byste se sami sebe: „Kde je `includeTeardownPages`, `includeSuiteTeardownPage` a `includeTeardownPage`?“ Což odpovídá tomu „...co jste do značné míry předpokládali.“

## Argumenty funkcí

Ideální počet argumentů funkce je nula. Jako další v počtu následuje jeden (monadická funkce), těsně následován dvěma (dyadická funkce). Třem argumentům (triadická) je třeba se vyhnout, kdykoliv je to možné. Počet více než tří (polyadicke funkce) vyžaduje důkladné zdůvodnění – a pak by se stejně neměl používat.

Argumenty nejsou jednoduchou záležitostí. Vyžadují mnoho konceptní energie. To je důvodem, proč jsem se v našem příkladu téměř všech zbavil. Podívejte se například ve vzorovém kódu na proměnnou `StringBuffer`. Mohli bychom ji předávat jako argument a ji



nevytvářet jako instanční proměnnou, ale pak by si ji naši čtenáři během každého čtení museli nějak interpretovat. Když čtete sdělení, které vám poskytuje modul, je název `includeSetupPage()` srozumitelnější než `includeSetupPageInto(newPageContent)`. Argument leží v jiné úrovni abstrakce než jméno funkce a nutí vás znát podrobnosti neboli `StringBuffer`, který v daném okamžiku není nijak důležitý.

Argumenty jsou těžkým oříškem i z pohledu testování. Představte si problémy při psaní všech modelových případů, abyste se ujistili, že všechny možné kombinace argumentů fungují správně. Pokud žádné nemáme, je to triviální. Pokud máme jeden argument, není to tak obtížné. Se dvěma argumenty je problém o něco horší. S více než se dvěma argumenty může být testování všech kombinací možných hodnot dost sklívající.

Výstupní argumenty jsou hůře srozumitelné než argumenty vstupní. Když čteme kód funkce, jsme zvyklí uvažovat tak, že informace vstupuje do funkce pomocí argumentů a vystupuje ven jako návratová hodnota. Obvykle neočekáváme, že by informace pomocí argumentů vystupovala ven. Takže výstupní argumenty nám mohou způsobovat dvojitou práci. Jeden vstupní argument je druhou nejlepší možností hned po variantě bez argumentů. Název `SetupTeardownIncluder.render(pageData)` je celkem lehce srozumitelný. Je jasné, že budeme získávat data v objektu `pageData`.

## Běžné tvary funkce s jedním argumentem

Existují dva velmi obecné důvody, proč funkci předávat jen jeden argument. Tímto argumentem se můžete dotazovat, jako například ve funkci `boolean fileExists("MyFile")`. Nebo jej můžete zpracovávat a transformovat na něco jiného a pak jej vrátit. Například funkce `InputStream fileOpen("MyFile")` transformuje název souboru typu `String` na návratovou hodnotu typu `InputStream`. Když čtenáři čtou kód funkce, očekávají tato dvě použití. Měli byste používat jména, která jasně vymezují rozdíl, a vždy používejte tyto dva tvary v konzistentním kontextu. (Viz níže stať Oddělování příkazů a dotazů.)

Poněkud méně obvyklým, ale stále velmi užitečným tvarem funkce s jedním argumentem je *událost*. U této formy existuje nějaký vstupní argument, ale neexistuje výstupní argument. Program má interpretovat volání této funkce jako událost a použít její argument tak, aby změnil stav systému. Například `void passwordAttemptFailedNtimes(int attempts)`. Tento tvar používejte opatrně. Čtenář by měl jasné vědět, že jde o událost. Volte opatrně jména i kontext.

Zkuste se vyhnout jakýmkoliv funkcím s jedním argumentem, které se těmto pravidlům vymykají, jako například `void includeSetupPageInto(StringBuffer pageText)`. Používání výstupního argumentu místo návratové hodnoty za účelem nějaké transformace je matoucí. Pokud funkce mění svoje vstupní argumenty, mělo by se to objevit v návratové hodnotě. Rozhodně je lepší `StringBuffer transform(StringBuffer in)` než `void transform(StringBuffer out)`, i když implementace prvního případu vrací jednoduchým způsobem vstupní argument. Alespoň stále vyhovuje formě transformace.

## Logické argumenty

Logické argumenty jsou špatné. Předávání logické proměnné funkci je opravdu strašný postup. Bezprostředně komplikuje signaturu metody a hlasitě volá, že tato funkce dělá více než jednu věc. Když je její hodnotou pravda, dělá jednu věc, a když to nepravda, dělá něco jiného!

Ve výpisu 3.7 jsme neměli možnost volby, protože volající objekty již logickou hodnotu předávaly a já jsem chtěl omezit rozsah refaktorování na úroveň funkce a kódu pod ní. Přesto je pro slabého čtenáře volání metody `render(true)` pouhým matením. Když budeme slídit kolem dokola, zjistíme, že tvar `render(boolean isSuite)` je o něco lepší, ale ne příliš. Měli bychom funkci rozdělit na dvě: `renderForSuite()` a `renderForSingleTest()`.

## Funkce se dvěma argumenty

Funkce se dvěma argumenty (dyadická) je hůř pochopitelná než funkce s jedním argumentem. Například funkce `writeField(name)` je srozumitelnější než `writeField(output_Stream, name)`.<sup>10</sup>

I když je význam obou alternativ jasný, když tu první přelétneme okem, uloží se nám její význam snadno do paměti. Druhá alternativa vyžaduje krátkou pauzu, dokud se nenaučíme první parametr ignorovat. A to může nakonec samozřejmě vést k problémům, protože žádnou část kódu bychom nikdy neměli ignorovat. Část kódu, kterou ignorujeme, je místo, kde se mohou skrývat chyby.

Někdy jsou samozřejmě dva argumenty opodstatněné. Například příkaz `Point p = new Point(0,0)` je zcela odůvodnitelný. Kartézské souřadnice zcela přirozeně potřebují dva argumenty. Určitě bychom byli velmi překvapeni, kdybychom viděli `new Point()`. Avšak dva argumenty jsou v tomto případě uspořádané složky jediné hodnoty! Na druhé straně `output_Stream` a `name` nemají ani přirozenou souvislost, ani přirozené uspořádání.

Dokonce i dyadické funkce, jako je `assertEquals(expected, actual)`, jsou problematické. Kolikrát jste vložili `actual` tam, kde mělo být `expected`? Tyto dva argumenty nemají žádné přirozené uspořádání. Pořadí `expected` a `actual` je konvencí, kterou se musíte naučit praxí.

Funkce se dvěma argumenty nejsou takovým zlem a určitě je někdy budete muset napsat. Ale měli byste mít na paměti, že to bude něco stát a že byste měli využít výhody dostupných mechanismů, kterými je možné je konvertovat na funkce s jedním argumentem. Můžete vytvořit například metodu `writeField` jako složku objektu `outputStream`, takže můžete napsat `outputStream.writeField(name)`. Nebo můžete vytvořit objekt `outputStream` jako členskou proměnnou aktuální třídy, takže jej nemusíte předávat. Nebo můžete extrahovat novou třídu, jako například `FieldWriter`, která přebírá objekt `outputStream` v konstruktoru a má metodu `write`.

## Funkce se třemi argumenty

Funkce se třemi argumenty (triadické) jsou mnohem hůř srozumitelné než funkce se dvěma argumenty. Otázky uspořádání, pauzy a ignorování jsou více než dvakrát komplikovanější. Doporučuji být velmi opatrný, než takovou funkci vytvoříte.

Podívejme se například na běžné přetížení funkce `assertEquals`, které přebírá tři argumenty: `assertEquals(message, expected, actual)`. Kolikrát jste četli `message` a mysleli si, že šlo o `expected`? Já jsem již mnohokrát u této konkrétní triadické funkce narazil a zastavil se. Ve skutečnosti *pokaždé, když ji vidím*, dlouze přemýšlím a pak si uvědomím, že parametr `message` mám ignorovat.

10. Právě jsem skončil s refaktorováním modulu, který používá dyadickou formu. Byl jsem schopen zavést `outputStream` jako pole třídy a převést veškerá volání metody `writeField` do jednoargumentové podoby. Výsledek byl mnohem jasnější.

Na druhé straně zde máme triádu, která není tak docela zákeřná: `assertEquals(1.0, amount, .001)`. I když to stále vyžaduje více přemýšlení, v tomto případě se to vyplatí. Vždy je dobré mít na paměti, že rovnost dvou hodnot v pohyblivé čárce je relativní záležitost.

## Objekty jako argumenty

Když se zdá, že funkce potřebuje více než dva nebo tři argumenty, je vhodné některé z nich zabalit do vlastní třídy. Uvažujte například o tom, jaký je rozdíl mezi následujícími dvěma deklaracemi:

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

Snížení počtu argumentů tím, že z nich vytvoříme objekt, může vypadat jako švindl, ale není tomu tak. Když se předávají společně skupiny proměnných, jako jsou ve výše uvedeném příkladu předávány parametry `x` a `y`, jsou pravděpodobně částí nějakého pojmu, který si zasluzuje samostatný název.

## Seznamy argumentů

Někdy si přejeme předat funkci variabilní počet argumentů. Podívejme se například na metodu `String.format`:

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

Mají-li se všechny argumenty zpracovat tímto způsobem, jako jsou zpracovány ve výše uvedeném příkladu, pak jsou ekvivalentní jednomu argumentu typu `List`. Podle této úvahy je metoda `String.format` vlastně dyadicá. A skutečně, deklarace metody `String.format`, jak je uvedeno níže, určitě dyadicá je.

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

Takže pro všechny se používá stejně pravidlo. Funkce, které mají proměnný počet argumentů, mohou být jedno-, dvoj- nebo dokonce tříargumentové. Ale bylo by chybou jim dávat více argumentů.

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

## Slovesa a klíčová slova

Výběr dobrého jména funkce může mít blízko k vysvětlování účelu funkce a k uspořádání a účelu argumentů. V případě jednoargumentové funkce by měla funkce spolu s argumentem tvořit nějakou velmi hezkou dvojici slovesa a podstatného jména. Například dvojice `write(name)` je velmi evokující. Ať už „name“ znamená cokoliv, je to něco, co je „zapisováno“. Ještě lepší kombinací může být `writeField(name)`, což nám říká, že „name“ je „field“ – datová složka.

Tento poslední případ je příkladem tvaru *klíčového slova* pro název funkce. Při používání tohoto tvaru kódujeme jména argumentů do názvu funkce. Například lze metodu `assertEquals` napsat lépe jako `assertExpectedEqualsActual(expected, actual)`. To značně zmírňuje problém, jak si zapamatovat pořadí argumentů.

## Žádné vedlejší efekty

Vedlejší efekt je klamem. Vaše funkce slibuje, že bude provádět jednu věc, ale dělá zároveň něco jiného, skrytého. Někdy provede neočekávané změny proměnných ve své vlastní trídě. Někdy je provede na parametrech předávaných funkci nebo na systémových globálních proměnných. Ve všech případech jsou tyto změny nevyzpytatelné a škodlivé klamy, které často vyústí v dočasné vazby a závislosti na pořadí.

Podívejte se například na zdánlivě neškodnou funkci ve výpisu 3.6. Tato funkce používá standardní algoritmus na shodu proměnných `userName` a `password`. Při shodě vrací hodnotu `true` a když něco není v pořádku, vrací hodnotu `false`. Ale má také vedlejší efekt. Naleznete jej?

### Výpis 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 ("Valid Password".equals(phrase)) {
                Session.initialize();
                return true;
            }
        }
        return false;
    }
}
```

Vedlejším efektem je samozřejmě volání metody `Session.initialize()`. Funkce `checkPassword` ve svém názvu říká, že kontroluje heslo. Z názvu ale neplynne, že inicializuje relaci. Takže ten, kdo ji volá a věří tomu, co říká její název, riskuje smazání dat z aktuální relace, když se rozhodne zkontrolovat platnost uživatele.

Tento vedlejší efekt vytváří časově závislou vazbu. To znamená, že `checkPassword` lze volat jen v určitý okamžik (jinými slovy jen tehdy, je-li inicializace relace bezpečná). Pokud není volána podle tohoto pravidla, mohou být data relace neúmyslně ztracena. Časově závislé vazby jsou matoucí, zvláště když jsou skryty v podobě vedlejších efektů. Pokud se bez dočasné vazby neobejdete, měli byste to v názvu funkce dát jasněj. V tomto případě bychom mohli přejmenovat funkci `checkPasswordAndInitializeSession`, přestože určitě porušuje pravidlo „dělej jednu věc“.

## Výstupní argumenty

Argumenty jsou nejpřirozeněji brány jako *vstupy* pro funkce. Pokud máte za sebou alespoň několik let programování, jsem si jist, že jste se pozastavili nad argumentem, který byl ve skutečnosti argumentem výstupním, nikoli vstupním. Například:

```
appendFooter(s);
```

Připojí tato funkce parametr `s` někam jako zápatí? Nebo připojí nějaké zápatí k parametru `s`? Je s nějaký vstup, nebo výstup? Při krátkém pohledu na signaturu funkce uvidíme:

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

To nám otázku zodpoví, ale jen za cenu kontroly deklarace funkce. Cokoliv, co vás nutí kontrolovat signaturu funkce, znamená zdržení. Je to zastávka za účelem poznání a měli byste se jí vyhýbat.

V dobách před objektově orientovaným programováním bylo někdy nezbytné mít výstupní argumenty. V objektově orientovaných jazycích se však potřeba výstupních argumentů značně snížila, protože jako výstupní parametr má fungovat `this`. Jinými slovy, bylo by lepší volat `appendFooter` jako `report.appendFooter()`;

Obecně bychom se měli výstupním argumentům vyhnout. Pokud musí vaše funkce měnit nějaký stav, ať změní stav svého vlastního objektu.

## Oddělování příkazů a dotazů

Funkce by měly buď něco dělat, nebo na něco odpovídat, ale neměly by dělat obojí. Bud'by vaše funkce měla měnit stav objektu, nebo by o tomto objektu měla dávat nějakou informaci. Bude-li dělat obojí, povede to k nedorozuměním. Podívejte se například na následující funkci:

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

Tato funkce nastavuje hodnotu pojmenovaného atributu a vrací hodnotu `true` po úspěšné operaci a hodnotu `false`, když žádný takový atribut neexistuje. To vede k podivnému příkazu, jako je tento:

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

Představte si to z úhlu pohledu čtenáře. Co to znamená? Ptá se na to, zda byl atribut „username“ dříve nastaven na „unclebob“? Nebo se ptá, zda byl atribut „username“ úspěšně nastaven na „unclebob“? Je těžké usuzovat na význam z volání funkce, protože není jasné, zda slovo „set“ je sloveso nebo přídavné jméno.

Autor měl v úmyslu, aby `set` bylo sloveso, ale z kontextu příkazu `if` se zdá, že jde o přídavné jméno, takže se příkaz čte jako „jestliže byl dříve atribut `username` nastaven na `unclebob` a ne „nastav atribut `username` na `unclebob`, a když to bude fungovat, pak...“ Mohli bychom to vyřešit přejmenováním funkce `set` na `setAndCheckIfExists`, ale v čitelnosti příkazu `if` nám to příliš nepomůže. Opravdové řešení je oddělit příkaz od dotazu, aby k této dvojznačnosti nemohlo dojít.

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

## Dejte přednost výjimkám před vracením chybových kódů

Vracení chybových kódů z příkazových funkcí je drobné porušení pravidla oddělování příkazů a dotazů. Podporuje to použití příkazů jako výrazů v predikátech příkazů `if`.

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

To nezpůsobuje nejasnost, zda jde o sloveso nebo přídavné jméno, ale vede ke hluboko vnořeným strukturám. Když vrátíte chybový kód, vytváříte problém, který spočívá v tom, že se volající musí chybou okamžitě zabývat.

```
if (deletePage(page) == E_OK) {  
    if (registry.deleteReference(page.name) == E_OK) {  
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK){  
            logger.log("page deleted");  
        } else {  
            logger.log("configKey not deleted");  
        }  
    } else {  
        logger.log("deleteReference from registry failed");  
    }  
} else {  
    logger.log("delete failed");  
    return E_ERROR;  
}
```

Na druhé straně, když použijete místo vracených chybových kódů výjimky, můžete kód pro zpracování chyb oddělit od bezchybné cesty kódu a lze jej zjednodušit:

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

## Extrahujte bloky Try/Catch

Bloky try/catch jsou již v samé podstatě nebezpečné. Zamlžují strukturu kódu a směšují zpracování chyb s normálním během programu. Je tedy lepší extrahovat těla bloků try/catch do samostatných funkcí.

```
public void delete(Page page) {  
    try {  
        deletePageAndAllReferences(page);  
    }  
    catch (Exception e) {  
        logError(e);  
    }  
}  
private void deletePageAndAllReferences(Page page) throws Exception {  
    deletePage(page);  
    registry.deleteReference(page.name);  
    configKeys.deleteKey(page.name.makeKey());  
}  
private void logError(Exception e) {  
    logger.log(e.getMessage());
```

Ve výše uvedeném kódu je funkce `delete` určena výlučně pro zpracování chyb. Je snadno srozumitelná a poté ji lze ignorovat. Funkce `deletePageAndAllReferences` se týká výlučně procesů, které kompletne mažou page. Zpracování chyb můžeme ignorovat. Zde vidíme dobře provedenou separaci, která usnadňuje chápání a modifikování kódu.

## Zpracování chyb je jedna věc

Funkce by mely dělat jednu věc. Zpracování chyb je jednou věcí. Když tedy funkce zpracovává chyby, neměla by dělat něco jiného. To naznačuje (jako ve výše uvedeném příkladu), že pokud ve funkci máte klíčové slovo `try`, mělo by být uvedeno jako první slovo této funkce a za bloky `catch`/`finally` by již nemělo nic následovat.

## Magnet závislosti Error.java

Vracení chybových kódů obvykle znamená, že existuje nějaká třída nebo výčtový typ, ve kterém jsou všechny tyto chybové kódy definovány.

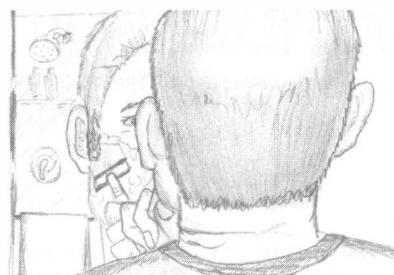
```
public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Takové třídy jsou *magnety závislosti*; mnoho jiných tříd je musí importovat a používat. Když se takto změní `Error enum`, je nutné všechny tyto třídy znova přeložit a instalovat.<sup>11</sup> To vytváří na třídu `Error` negativní tlak. Programátoři nechtějí přidávat nové chyby, protože pak musejí vše znova sestavovat a instalovat. Takže místo přidávání nových raději používají staré chybové kódy.

Používáte-li místo chybových kódů výjimky, jsou nové třídy výjimek *odvozeny* od třídy stávajících tříd výjimek. Nové třídy lze přidávat bez nutnosti jakéhokoliv překladu nebo nové instalace.<sup>12</sup>

## Neopakujte se<sup>13</sup>

Podívejte se znova pečlivě na výpis 3.1 a všimněte si, že je tam algoritmus, který se opakuje čtyřikrát – pro každý případ `SetUp`, `SuiteSetUp`, `TearDown` a `SuiteTearDown`. Není jednoduché toto zdvojení rozpoznat, protože tyto čtyři instance jsou promíchány s jiným kódem a nejsou opakovány úplně stejně. Přesto je opakování problémem,



11. Našli se i takoví, kteří měli pocit, že se obejdou bez nového překladu a instalace, a podle toho se s nimi jednalo.

12. To je příklad principu otevřenosti a uzavřenosti (OCP) [PPP02].

13. Princip DRY (Don't Repeat Yourself) [PRAG].

protože nafukuje kód a při změně algoritmu bude vyžadovat modifikaci na čtyřech místech. Je to také čtyrnásobná možnost něco vynechat.

Toto zdvojení bylo ošetřeno metodou `include` ve výpisu 3.7. Pročtěte si tento kód ještě jednou a všimněte si, nakolik se odstraněním tohoto zdvojení zlepšila čitelnost celého modulu.

Opakování může být v softwaru kořenem všeho zla. Pro jeho odstranění nebo omezení bylo vytvořeno mnoho principů a postupů. Podívejte se například na to, že všechny Coddovy databázové normální formy slouží k eliminaci opakování dat. Podívejte se i na to, jak umožňuje objektově orientované programování soustředit kód do základních tříd, které by jinak byly redundantní. Strukturované programování, aspektově orientované programování nebo komponentově orientované programování – to jsou všechno strategie, které se snaží částečně eliminovat zdvojení. Zdá se, že od objevení podprogramu spočívají inovace ve vývoji softwaru v neustálých pokusech o eliminaci opakování zdrojového kódu.

## Strukturované programování

Někteří programátoři se řídí pravidly strukturovaného programování, jejichž autorem je Edsger Dijkstra.<sup>14</sup> Dijkstra prohlásil, že každá funkce a každý blok v rámci nějaké funkce by měly mít jeden vstupní a jeden výstupní bod. Podle těchto pravidel by měl být v jedné funkci jen jeden příkaz `return`, cyklus by neměl obsahovat žádné příkazy `break` nebo `continue` a skutečně nikdy žádný příkaz `goto`.

I když podporujeme cíle a postupy strukturovaného programování, u velmi malých funkcí nepřináší tato pravidla příliš mnoho užitku. Výrazný užitek přináší jen u velkých funkcí.

Jsou-li vaše funkce krátké, příležitostné použití vícenásobného příkazu `return`, `break` nebo `continue` nenadělá příliš mnoho škody a může být naopak výstižnější než pravidlo jednoho vstupu a jednoho výstupu. Na druhé straně příkaz `goto` má smysl pouze ve velkých funkčích a měli bychom se mu vyhnout.

## Jak napíšete funkci, jako je tato?

Psaní softwaru je psaní jako každé jiné. Když píšete přednášku nebo nějaký článek, nejdříve si zapíšete své myšlenky a pak text masírujete tak dlouho, dokud není dobře čitelný. První pracovní verze může být neohrabaná a neusporyádaná, a tak ji vylepšujete a restrukturalizujete a pročistíujete, dokud není čitelná tak, jak si to přejete.

Když píšu funkce, bývají dlouhé a komplikované. Mají mnoho odsazování a vnořených cyklů. Mají také dlouhý seznam argumentů. Jména jsou náhodná a kód se opakuje. Ale mám i sadu jednotkových testů, které pokrývají všechny řádky tohoto neohrabaného kódu.

Takže tento kód masíruji a pročistíuji, rozděluji funkce, měním jména, odstraňuji opakování. Zkracuji funkce a reorganizuji je. Někdy rozdělím celé třídy, vše za neustálého testování.

Nakonec se dopracuji k funkcím, které vyhovují pravidlům, o nichž jsme v této kapitole mluvili. Tímto způsobem je nepíšu na počátku. Nemyslím, že by to kdokoliv dokázal.

14. [SP72].

## Závěr

Každý systém je sestaven z doménově specifického jazyka, navrženého programátory tak, aby systém popisoval. Funkce jsou slovesy tohoto jazyka a třídy jsou podstatnými jmény. To není nějaký návrat k nehezkému zápisu, ve kterém jsou v zadávacím dokumentu podstatná jména a slovesa prvním nástinem tříd a funkcí systému. Spiše se jedná o mnohem starší skutečnost. Umění programovat je a vždy bylo uměním návrhu jazyka.

Odborníci na programování berou systém jako příběh, který se má vyprávět, a ne jako program, jenž se má napsat. Používají možnosti jazyka, který si zvolí, aby zkonstruovali mnohem bohatší a expresivnější jazyk, který lze pro vyprávění tohoto příběhu použít. Částí tohoto doménově specifického jazyka je hierarchie funkcí, která popisuje veškeré činnosti, k nimž v systému dochází. Během tohoto důmyslného a rekurzivního postupu jsou tyto činnosti napsány tak, aby využívaly doménově specifický jazyk, formulovaný tak, aby vyprávěl svou malou část tohoto příběhu.

Tato kapitola pojednávala o mechanismech dobrého psaní funkcí. Pokud se budete těmito pravidly řídit, budou vaše funkce krátké, dobře pojmenované a hezky organizované. Ale nikdy nezapomínejte, že váš skutečný cíl je vyprávět příběh systému a že funkce, které píšete, musejí k sobě čistě pasovat a vytvářet srozumitelný a precizní jazyk, jenž vám v tomto vyprávění pomůže.

### Výpis 3.7. SetupTeardownIncluder.java

```
package fitness.html;
import fitness.responders.run.SuiteResponder;
import fitness.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");
}
}
```

## Použitá literatura

[KP78]: Kernighan and Plaugher, *The Elements of Programming Style*, 2. vydání, McGraw-Hill, 1978.

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

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

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

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

# KAPITOLA 4

## Komentáře

### V této kapitole najdete:

- ◆ Komentáře nevyváží špatný kód
- ◆ Vyjádřete se kódem
- ◆ Dobré komentáře
- ◆ Špatné komentáře
- ◆ Literatura



„Nedávejte komentáře do špatného kódu – přepište jej.“

Brian W. Kernighan a P. J. Plaugher<sup>1</sup>

Není nic užitečnějšího než komentář na správném místě. Nic nemůže zaneřádit modul více než zbytečné a dogmatické komentáře. Nic nemůže škodit více než zastarálý a špatný komentář, který klame a dezinformuje.

Komentáře nejsou Schindlerovým seznamem. Nejsou „čistým dobrem“. Ve skutečnosti jsou komentáře v nejlepším případě nutným zlem. Pokud je náš programovací jazyk dostatečně expresivní nebo pokud máme talent k jejich rozumnému užití a vyjádření svého záměru, komentáře bychom příliš používat nemuseli – možná vůbec.

Skutečným důvodem pro jejich používání je kompenzace našeho neúspěchu vyjádřit se pomocí kódu. Všimněte si, že jsem použil slovo *neúspěch*. Myslel jsem to vážně. Komentáře jsou vždy neúspěchem. Musíme je mít, protože ne vždy dokážeme přijít na to, jak se vyjádřit bez nich, ale jejich použití není důvodem k oslavě.

Jestliže se tedy nacházíte v situaci, kdy potřebujete napsat komentář, promyslete si to a zvažte, zda by neexistoval nějaký jiný způsob a zda by nebylo možné se vyjádřit v kódu. Pokaždé, když se v kódu umíte vyjádřit, měli byste se poplácat na zádech. Pokaždé, když napíšete komentář, byste se měli ušklíbnout a pocítit neúspěch svých vyjadřovacích schopností.

Proč jsem na komentáře tak příkrý? Protože neříkají pravdu. Příliš často, i když ne vždycky a ne záměrně. Čím je komentář starší a čím je umístěn dále od kódu, který popisuje, tím je větší pravděpodobnost, že bude prostě špatně. Důvod je jednoduchý. Není reálné, že by je programátoři dokázali udržovat a aktualizovat.

Kód se mění a vyvíjí. Jeho části se přesunují z místa na místo. Tyto části se rozvětvují a reprodukují a opět dávají dohromady, aby se zformovaly v přízraky. Naneštěstí přesun kódu neznamená vždy přesun komentáře – a ani to *není možné*. A tak jsou komentáře příliš často oddělovány od kódu, který popisují, a stávají se z nich osiřelé upoutávky menší a menší přesnosti. Podívejte se, co se například stalo tomuto komentáři na řádku, který měl popisovat:

```
MockRequest request;
private final String HTTP_DATE_REGEX = 
"[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;
// Příklad: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Později byly pravděpodobně přidány nějaké instanční proměnné a byly vloženy mezi konstantu `HTTP_DATE_REGEX` a vysvětlující komentář.

Zde je možné udělat závěr, že by programátoři měli být natolik disciplinovaní, aby udržovali komentáře v bezvadném stavu, platné a precizní. Souhlasím, měli by. Ale raději bych na prvním místě investoval tuto energii do takové čitelnosti a expresivnosti kódu, který by komentáře nepotřeboval.

1. [KP78], p. 144.

Nepřesné komentáře jsou mnohem horší než žádné. Jsou klamavé a zavádějící. Vytvářejí očekávání, která nebudou nikdy splněna. Stanovují stará pravidla, která nepotřebujeme nebo jež bychom již nikdy neměli používat.

Skutečnost bychom měli objevovat jen na jednom místě: v kódu. Jedině kód vám může doopravdy říci, co dělá. Je jediným zdrojem opravdové a precizní informace. Takže ačkoliv jsou někdy komentáře nutné, vyvineme značné úsilí, aby bychom je minimalizovali.

## Komentáře nevyváží špatný kód

Jedním z nejobecnějších motivací pro psaní komentářů je špatný kód. Píšeme modul a víme, že je zmatený a neusporádaný. Víme, že je to chaos. Takže si řekneme: „Ó, bude lepší, když to okomentuj!“ Nikoliv! Raději to vyčistěte!

Jasný a expresivní kód s několika komentáři je mnohem kvalitnější, než je zmatený a komplikovaný kód s mnoha komentáři. Spíše venujte váš čas odstranění nepořádku, než byste psali komentáře, které by ho vysvětlovaly.

## Vyjádřete se kódem

Určitě existují případy, které nelze za pomocí kódu dobře vysvětlit. Bohužel mnoho programátorů tento názor přijalo s tím, že kód je málokdy, jestli vůbec, dobrým vyjadřovacím prostředkem. To je očividně špatně. Co byste viděli raději? Tohle:

```
// Zkontrolujte, zda má zaměstnanec nárok na plný rozsah výhod
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

Nebo tohle?

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

Na to, abyste vyjádřili v kódu podstatnou část svého záměru, stačí jen několik sekund přemýšlení. V mnoha případech je to prostě jen otázkou vytvoření funkce, která sdělí totéž, co komentář, jež chcete napsat.

## Dobré komentáře

Komentáře jsou někdy nutné nebo prospěšné. Podíváme se na některé z nich, které stojí za pozornost. Mějte ale na paměti, že jediným skutečně dobrým komentářem je ten, u kterého jste našli způsob, jak se bez něj obejít.

## Komentáře právnického charakteru

Někdy vás standardy pro kódování ve vaší firmě nutí napsat některé poznámky z právnických důvodů. Například sdělení o autorských právech jsou nutná a zdůvodnitelná, protože je máte umístit na začátek každého zdrojového kódu.

Zde máme například standardní záhlaví s komentáři, které umisťujeme na začátku každého zdrojového souboru v nástroji FitNesse. Jsem rád, že mohu říci, že naše integrované vývojové prostředí tento komentář automaticky potlačí, aby nezpůsoboval zbytečné zaneřádění kódu.

```
// Copyright (C) 2003,2004,2005; Object Mentor, Inc. Veškerá práva vyhrazena.  
// Vydáno podle podmínek GNU (General Public License) verze 2 nebo pozdější.
```

Komentáře tohoto druhu by neměly mít charakter smlouvy nebo objemných právnických knih. Umisťujte odkazy na standardní licence nebo jiné externí dokumenty všude tam, kde je to možné a nedávejte do komentářů veškeré a úplné požadavky a podmínky.

## Informativní komentáře

Někdy je užitečné poskytnout v komentáři základní informace. Podívejte se například na tento komentář, který vysvětluje návratovou hodnotu abstraktní metody:

```
// Vrací instanci typu Responder, která je testovaná.  
protected abstract Responder responderInstance();
```

Komentář tohoto druhu může být někdy užitečný, ale pokud je to možné, je lepší používat ke sdělení informace název funkce. V tomto případě by například mohla být poznámka zbytečná, kdybychom přejmenovali funkci responderBeingTested.

Tohle je o něco lepší:

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

Zde nám komentář sděluje, že regulární výraz má odpovídat času a datu, které byly zformátovány pomocí funkce SimpleDateFormat.format za použití specifikovaného formátovacího řetězce. Bylo by však asi lepší a čistější, kdyby byl tento kód přemístěn do zvláštní třídy, která by konvertovala formáty datových a časových údajů. Pak by mohl být komentář nadbytečný.

## Vysvětlení záměru

Komentář může někdy jít i za hranice běžné informace o implementaci a může sdělovat úmysl nějakého rozhodování. V následujícím případě můžeme vidět zajímavé rozhodování popsané v komentáři. Během porovnávání dvou objektů se autor rozhodl, že bude objekty této třídy řadit výše než objekty kterékoli jiné třídy.

```
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; // jsme větší, protože jsme ten správný typ.  
}
```

Tady je dokonce lepší příklad. Nemusíte souhlasit s programátorovým řešením tohoto problému, ale aspoň víte, o co se pokoušel.

```
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);
    //Tohle je náš nejlepší pokus získat konflikt časování tím,
    //že vytvoříme velký počet vláken.
    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());
}
```

## Objasnění

Někdy je prostě užitečné přeložit význam nějakého obskurního argumentu nebo návratové hodnoty do nějaké čitelné formy. Obecně je lepší nalézt způsob, jak udělat argument nebo návratovou hodnotu srozumitelnou samu o sobě. Ale pokud je součástí standardní knihovny nebo kódu, který nemůžete měnit, pak může být vysvětlující pomocný komentář užitečný.

```
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
}
```

Existuje samozřejmě podstatné riziko, že vysvětlující komentář nebude správný. Projděte si předchozí příklad a podívejte se, jak obtížné je ověřit si jejich správnost. To vysvětluje jednak, proč je vysvět-

lení nezbytné a jednak, proč je i riskantní. Proto před psaním podobných komentářů dobře zvažte, zda neexistuje nějaký lepší způsob, a věnujte ještě větší péči jejich preciznosti.

## Varování před důsledky

Někdy je užitečné varovat jiné programátory před určitými důsledky. Zde například máme komentář, který vysvětluje, proč je určitý test vypnutý:

```
// Nespouštějte, pokud nemáte nějaký čas na
// odstřelení procesu.
public void _testWithReallyBigFile()
{
    writeLinesToFile(10000000);
    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertEquals("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```



Dnes bychom samozřejmě test vypnuli za použití atributu @Ignore s příslušným vysvětlujícím řetězcem. @Ignore("Takes too long to run"). Ale v dobách ještě před vznikem JUnit 4 bylo běžným zvykem umístit před název metody podržítko. Tento komentář, ač prostořeký, velmi dobře vystihuje pointu.

Zde máme další příklad:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    //SimpleDateFormat není zabezpečený pro podprocesy,
    //takže každou instanci musíme vytvořit nezávisle.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}
```

Můžete namítat, že existují lepší způsoby, jak tento problém vyřešit, a já bych s vámi mohl souhlasit. Ale komentář, tak jak je zde podán, je maximálně racionální. Zabrání některým příliš dychtivým programátorům použít v zájmu efektivity statický inicializátor.

## Komentáře TODO (co dělat)

Někdy je rozumné zanechat poznámky „co dělat“ ve formě komentářů //TODO. V následujícím příkladě vysvětuje komentář typu „co dělat“, proč má funkce degenerovanou implementaci a co by měla v budoucnu obsahovat.

```
// TODO-MdM nejsou zapotřebí
// Předpokládáme, že to vyřadíme po vytvoření přezkušovacího modelu
protected VersionInfo makeVersion() throws Exception
{
    return null;
}
```

„Co dělat“ jsou práce, o kterých si programátor myslí, že by se měly provést, ale z nějakého důvodu to není možné právě teď. Může to být připomínkou ke smazání nějaké nepotřebné vlastnosti nebo žádost, aby se někdo na problém podíval. Může to být požadavek na někoho jiného, aby vymyslel lepší název, nebo připomínka udělat změnu, která je závislá na nějaké plánované události. Ať už je to cokoliv, poznámka „co dělat“ *není* omluvou pro to, aby špatný kód, zůstal v systému.

Většina integrovaných vývojových systémů poskytuje v dnešních dobách speciální podporu a nástroje pro nalezení veškerých komentářů typu „co dělat“, takže není pravděpodobné, že by se se ztratily. Přesto nechcete svůj kód těmito poznámkami zaneřádit, takže je pravidelně procházejte a odstraňujte ty, které odstranit lze.

## Zvýraznění

Komentář může být použit pro zvýraznění důležitosti něčeho, co se jinak může zdát bezvýznamným.

```
String listItemContent = match.group(3).trim();
// ořezání je opravdu důležité. Odstraňuje vedoucí
// mezery, které by mohly způsobit, že by položka byla považována
// za další seznam.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

## Javadoc ve veřejných API

Není nic užitečnějšího a prospěšnějšího než dobré popsané veřejné API. Reálným příkladem je nástroj Javadoc pro standardní knihovnu Java. Bez něj by bylo psaní programů v Java přinejmenším obtížné.

Pokud píšete veřejné API, pak byste určitě měli pro to napsat dokumentaci v nástroji Javadoc. Ale mějte na paměti druhou část této kapitoly. Tyto dokumenty mohou být také zavádějící, nemusejí se vztahovat k lokálnímu problému a mohou být klamné jako kterýkoliv jiný komentář.

## Špatné komentáře

Většina komentářů spadá do této kategorie. Obyčejně se jedná o berličky nebo omluvy špatného kódu nebo o ospravedlnění nedostatečných rozhodnutí, což je jen něco více než programátorova samomluva.

## Huhňání

Plácání v komentáři jen proto, že máte pocit, že se to má, nebo protože to proces vyžaduje, je pisákovství. Když se rozhodnete napsat komentář, věnujte tomu nezbytný čas a ujistěte se, že je to ten nejlepší komentář, kterého jste schopni.

Například zde je případ, který jsem nalezl ve FitNesse, kde by komentář mohl být opravdu užitečný. Ale autor spěchal nebo tomu nevěnoval dost pozornosti. Výsledkem jeho umění byla hádanka:

```

public void loadProperties()
{
    try
    {
        String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
        FileInputStream propertiesStream = new FileInputStream(propertiesPath);
        loadedProperties.load(propertiesStream);
    }
    catch(IOException e)
    {
        // Neexistují-li žádné soubory vlastností, jsou zavedeny implicitní
    }
}

```

Co znamená poznámka v bloku `catch`? Očividně měla jakýsi význam pro autora, ale naprosto není jasné čtenáři. Zřejmě platí, že pokud nastane výjimka typu `IOException`, znamená to, že soubor vlastností neexistuje. V tom případě jsou načteny všechny implicitní hodnoty. Ale kdo je nače? Kde se načítají před voláním metody `loadProperties.load`? Nebo tuto výjimku zachytává `loadProperties.load`, načež implicitní hodnoty a pak výjimku předá, abychom ji ignorovali? Nebo načež `loadProperties.load` všechny implicitní hodnoty ještě před tím, než se pokusí číst soubor? Nebyl autor příliš pohodlný, když zanechal blok `catch` prázdný? Nebo – a to je děsivá možnost – pokoušel se autor říci sám sobě, že se sem má vrátit a dopsat kód pro implicitní hodnoty později?

Naše jediné východisko spočívá v tom, že prozkoumáme kód v jiných částech systému a zjistíme, o co jde. Jakýkoliv komentář, jehož pochopení vás nutí ke zkoumání jiných modulů, vám nedokázal nic říci a nestojí zazlámanou grešli.

## Nadbytečné komentáře

Výpis 4.1 ukazuje jednoduchou funkci s komentářovým záhlavím, které je zcela nadbytečné. Čtení komentáře bude pravděpodobně delší než čtení samotného kódu.

### **Výpis 4.1. `waitForClose`**

```

// Pomocná metoda, která vrací hodnotu, má-li this.closed hodnotu true. Vyvolá
// výjimku, vyprší-li časový limit.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
    if(!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception("MockResponseSender could not be closed");
    }
}

```

K jakému účelu tento komentář slouží? Určitě neinformuje lépe než sám kód. Kód nevysvětluje a neposkytuje ani účel kódu nebo nějaké zdůvodnění. Jeho čtení není jednodušší než čtení kódu. Skutečně je méně přesný než vlastní kód a svádí čtenáře akceptovat tento nedostatek přesnosti místo toho, aby

se snažil o skutečné porozumění kódu. Je to spíše potrásání rukou prodejce ojetin, který vás ujišťuje, že není nutné nahlížet pod kapotu.

Podívejte se nyní na obrovský počet zbytečných a redundantních komentářů javadoc ve výpisu 4.2, vytvořený pomocí Tomcatu. Tyto komentáře slouží jen k zaneřádění a zatemnění kódu. Nemají naprosto žádný dokumentační účel. Horší je, že jsem vám ukázal jen několik prvních. V tomto modulu je jich mnohem více.

#### Výpis 4.2. ContainerBase.java (Tomcat)

```
public abstract class ContainerBase
    implements Container, Lifecycle, Pipeline,
    MBeanRegistration, Serializable {
    /**
     * Zpoždění procesoru pro tuto komponentu.
     */
    protected int backgroundProcessorDelay = -1;
    /**
     * Podpora životního cyklu události této komponenty
     */
    protected LifecycleSupport lifecycle =
        new LifecycleSupport(this);
    /**
     * Příjemci událostí tohoto kontejneru.
     */
    protected ArrayList listeners = new ArrayList();
    /**
     * Implementace zaváděče, se kterým je sdružen tento kontejner.
     */
    protected Loader loader = null;
    /**
     * Implementace protokolovacího nástroje, se kterým je asociován
     * tento kontejner.
     */
    protected Log logger = null;
    /**
     * Název sdruženého protokolovacího nástroje.
     */
    protected String logName = null;
    /**
     * Implementace správce, se kterým je sdružen tento kontejner.
     */
    protected Manager manager = null;
    /**
     * Skupina, se kterou je sdružen tento kontejner.
     */
    protected Cluster cluster = null;
    /**
     * Název tohoto kontejneru čitelný pro člověka.
     */
    protected String name = null;
    /**
```

```
* Nadřazený kontejner, kterému je tento kontejner podřízený.  
*/  
protected Container parent = null;  
/**  
 * Zavaděč nadřazené třídy, který se má konfigurovat během  
 * instalace zavaděče.  
 */  
protected ClassLoader parentClassLoader = null;  
/**  
 * Kanálový objekt, se kterým je sdružen tento kontejner.  
 */  
protected Pipeline pipeline = new StandardPipeline(this);  
/**  
 * Oblast, se kterou je sdružen tento kontejner.  
 */  
protected Realm realm = null;  
/**  
 * Zdroje objektu DirContext, se kterým je sdružen  
 * tento kontejner  
 */  
protected DirContext resources = null;
```

## Matoucí komentáře

Někdy ve svých poznámkách napíše programátor i s těmi nejlepšími úmysly sdělení, které není dost přesné. Podívejte se chvíliku na velmi redundantní, ale také poněkud matoucí komentář, který jsme viděli ve výpisu 4.1.

Zjistili jste, proč je tento komentář matoucí? Metoda nevrací hodnotu, *když* `this.closed` nabude hodnoty true. Vrací hodnotu, *jestliže* `this.closed` je true. Jinak čeká nějakou dobu a pak vyvolá událost, *jestliže* `this.closed` není stále true.

Tato jemná dezinformace, stylizovaná do komentáře, který se čte obtížněji než tělo kódu, může způsobit, že jiný programátor může slepě volat tuto funkci a očekávat, že obdrží návratovou hodnotu ihned, jakmile `this.closed` nabude hodnoty true. Tento nešťastný programátor pak bude během ladění zjišťovat, proč je jeho kód tak pomalý.

## Závazné komentáře

Je jednoduše hloupé mít pravidlo, které říká, že každá funkce musí mít komentář pro javadoc nebo že každá proměnná musí mít komentář. Podobné komentáře pouze zaplňují kód, šíří nepravdy a přispívají k celkovému zmatku a dezorganizaci.

Například vyžadování komentářů pro javadoc ke každé funkci vede takovým ohavnostem, jako je kód ve výpisu 4.3. Tento nepořádek nepřináší nic pozitivního, přispívá jen k zatemňování kódu a dává potenciálně prostor pro klamné informace a chybné nasměrování.

## Deníkové komentáře

Lidé někdy přidávají komentáře na začátek modulu pokaždé, když ho editují. Tyto komentáře se kumulují do jakéhosi deníku nebo protokolového souboru všech změn, které kdy byly provedeny. Viděl jsem některé moduly s mnoha stránkami podobných nesouvisejících deníkových zápisů.

**Výpis 4.3.** Deníkový komentář

```
/***
 *
 * @param title Titul CD
 * @param author Autor CD
 * @param tracks Počet stop na CD
 * @param durationInMinutes Doba trvání CD v minutách
 */
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);
}
```

Kdysi existovaly dobré důvody, proč deníkové záznamy na začátku každého modulu vytváret a udržovat. Neměli jsme systémy správu zdrojových kódů, který by to dělal za nás. Ale dnes slouží tyto dlouhé záznamy jen pro zaplňování a zatemňování modulu. Měly by být zcela vyřazeny.

**Komentáře obsahující šum**

Někdy uvidíte komentáře, které neobsahují nic jiného než šum. Opakují samozřejmě a neposkytují žádnou novou informaci.

```
/***
 * Implicitní konstruktor.
 */
protected AnnualDateRule() {
```

*Opravdu ne?* Nebo co třeba tohle:

```
/** Den v měsíci. */
private int dayOfMonth;
```

A zde je ideál nadbytečnosti:

```
/***
 * Vrací den v měsíci.
 *
 * @return den v měsíci.
 */
public int getDayOfMonth() {
    return dayOfMonth;
}
* Změny (od 11-10-2001)
* -----
* 11-Oct-2001 : Změna v organizaci třídy a přesun do nového balíčku
* com.jrefinery.date (DG);
```

- \* 05-Oct-2001 : Přidána metoda Added a getDescription(), a odstraněna třída
- \* NotableDate
- \* 12-Nov-2001 : IBD vyžaduje metodu setDescription(), když byla nyní třída
- \* NotableDate vyřazena; změna metod Changed getPreviousDayOfWeek(),
- \* getFollowingDayOfWeek() and getNearestDayOfWeek() za účelem opravy chyb;
- \* 05-Dec-2001 : Opravena chyba ve třídě SpreadsheetDate (DG);
- \* 29-May-2002 : Měsíční konstanty přesunuty do zvláštního rozhraní
- \* (MonthConstants) (DG);
- \* 27-Aug-2002 : Opravena chyba v metodě addMonths(), díky Petru Nálevkovi (DG);
- \* 03-Oct-2002 : Opraveny chyby podle výpisu nástroje Checkstyle (DG);
- \* 13-Mar-2003 : Serializable, implementováno(DG);
- \* 29-May-2003 : Opravena chyba v metodě addMonths (DG);
- \* 04-Sep-2003 : Comparable, implementováno. isInRange javadocs, aktualizováno(DG);
- \* 05-Jan-2005 : Operavena chyba v metodě addYears() (1096282) (DG);

Tyto komentáře obsahují také šumu, že si zvykneme je ignorovat. Během čtení kódu je naše oči prostě přeskocí. Nakonec tyto komentáře budou odpovídat pravdě tím méně, čím více se kód v jejich blízkosti bude měnit.

První komentář na výpisu 4.4 se zdá být vhodný.<sup>2</sup> Vysvětluje, proč je ignorován blok catch. Ale druhý komentář je čirý šum. Očividně byl programátor psaním boků try/catch natolik frustrován, že se potřeboval odreagovat.

#### Výpis 4.4. startSending

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normální. Někdo požadavek zastavil.
    }
    catch(Exception e)
    {
        try
        {
            response.add(ErrorResponder.makeExceptionString(e));
            response.closeAll();
        }
        catch(Exception e1)
        {
            //Dej mi trochu času!
        }
    }
}
```

---

2. Současný trend v moderních integrovaných vývojových systémech je kontrolovat pravopis v poznámkách bude bezzájem pro ty, kteří čtou velká množství kódu.

Programátor by měl pochopit, že může svou frustraci vyřešit spíše zlepšením struktury kódu, než se odreagovávat psaním bezcenných a rušivých komentářů. Měl svou energii přesměrovat k extrakci posledního bloku `try/catch` do samostatné funkce, jak ukazuje výpis 4.5.

**Výpis 4.5. startSending (po refaktorování)**

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normální. Někdo dotaz zastavil.
    }
    catch(Exception e)
    {
        addExceptionAndCloseResponse(e);
    }
}
private void addExceptionAndCloseResponse(Exception e)
{
    try
    {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    }
    catch(Exception e1)
    {
    }
}
```

Vystříhejte se pokušení vytvářet šum a raději se rozhodněte k vyčistění svého kódu. Zjistíte, že budeťte lepším a spokojenějším programátorem.

## Rušení nahánějící hrůzu

Komentáře pro javadocs mohou být také plné šumu. Jaký účel mají následující komentáře z dobře známé knihovny s otevřeným kódem? Odpověď zní, že žádný. Jsou to jen nadbytečné rušivé komentáře, napsané z nějaké pomýlené touhy poskytnout dokumentaci.

```
/** Název. */
private String name;
/** Verze. */
private String version;
/** Název licence. */
private String licenceName;
/** Verze. */
private String info;
```

Pročtěte si tyto komentáře ještě jednou a pozorněji. Vidíte chybu, vzniklou během vyjmutí a vložení textu? Pokud autoři nevěnují náležitou pozornost psaní komentářů (nebo jejich vkládání), proč bychom měli očekávat, že budou čtenáři k užitku?

## Nepoužívejte komentář, když můžete použít funkci nebo proměnnou

Podívejte se na následující úsek kódu:

```
// závisí tento modul z globálního seznamu <mod> na
// subsystému, jehož jsme součástí?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

To lze přeforumulovat bez komentářů takto:

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

Možná, že autor originálního kódu napsal nejdříve komentáře (nepravděpodobné) a poté kód, aby vyhověl komentářům. Avšak autor měl poté kód refaktorovat tak, jak jsem to udělal já, aby bylo možné komentáře odstranit.

## Označení pozice

Programátoři někdy rádi označují určitou pozici ve zdrojovém souboru. Například jsem nedávno našel v programu, který jsem procházel, tohle:

```
// Akce /////////////////////////////////
```

V některých řídkých případech má smysl dávat dohromady určité funkce pod poutač, jako je tenuhle. Ale obecně se jedná o nepořádek, který by měl být odstraněn – zvláště ten rušivý sled lomítek na konci řádku.

Berte to takto. Poutače jsou překvapivé a nápadné, pokud se s nimi nesetkáváte příliš často. Využijte je tedy s mírou a jen tehdy, budou-li opravdu přínosem. Pokud je budete používat nadměrně, stane se z nich šum na pozadí a čtenáři je budou ignorovat.

## Komentáře na konci složených závorek

Někdy programátoři umisťují speciální komentáře na konec složených závorek, jak ukazuje výpis 4.6. Ačkoliv u dlouhých funkcí s vícenásobně vnořenými strukturami to může mít smysl, do malých a zapouzdřených funkcí, kterým dáváme přednost, to vnáší jen nepořádek. Takže pokud máte nutkání okomentovat konec složených závorek, pokuste se raději zkrátit své funkce.

### Výpis 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
}
```

## Připisování a podtitulky se jmény

/\* Přidal Rick \*/

Systémy řízení zdrojových kódů jsou velmi užitečné pro zachování informace o tom, kdo co přidal a kdy. Není zapotřebí zanášet kód těmito drobnými podtitulky. Můžete si myslet, že by podobné komentáře mohly být užitečné, aby pomáhaly ostatním v tom, na koho se obracet ohledně kódu. Ale skutečnost je taková, že mají tendenci v textu setrvávat léta a léta a být čím dál tím méně přesné a relevantní.

Říkám znovu, že pro tento druh informací je lepším místem systém pro správu zdrojových kódů.

## Zakomentované řádky kódů

Jen málo praktik je tak špatných, jako je zakomentovaný kód. Nedělejte to!

```
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()));
```

Jiní, kteří tyto zakomentované řádky kódů čtou, nebudou mít odvahu je smazat. Budou si myslet, že pro to existuje důvod a že jsou příliš důležité, než aby mohly být smazány. Takže se zakomentovaný kód hromadí jako sedlina v lávci se špatným víнем.

Podívejte se na tento kód z komponent apache commons:

```
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;
```

Proč jsou zde dva řádky kódu zakomentovány? Jsou důležité? Byly zde ponechány jako připomínka pro nějakou nadcházející změnu? Nebo se pouze jedná o nadbytečný kód, který před lety někdo zakomentoval a prostě se nepostaral o jeho vyčistění?

V šedesátých letech byly časy, kdy zakomentování kódu mohlo být užitečné. Ale nyní máme již dlouho systémy pro správu zdrojových kódů. Tyto systémy si budou kód pamatovat za nás. Kód již nemůžeme zakomentovávat. Stačí jej vymazat. Neztratíme ho. Slibuji.

## Komentáře ve formátu HTML

Ve zdrojovém kódu je kód HTML ohavností, jak si můžete přečíst v kódu uvedeném níže. Způsobuje, že komentáře jsou hůř čitelné právě v tom místě, kde by se měly číst dobré – v editoru integrovaného vývojového prostředí. Pokud budou komentáře extrahovány pomocí nějakého nástroje (jako Javadoc), aby byly zobrazeny jako webová stránka, měla by to být funkčnost tohoto nástroje a ne programátora, aby komentáře zkrášloval příhodným kódem HTML.

```
/**
 * Úloha, která má spustit testy fit.
 * Tato úloha spouští testy kvality a publikuje výsledky.
 * <p/>
 * <pre>
 * Usage:
 * <taskdef name="execute-fitnessse-tests"
 * classname="fitnessse.ant.ExecuteFitnessseTestsTask"
 * classpathref="classpath" />
 * OR
 * <taskdef classpathref="classpath"
 * resource="tasks.properties" />
 * <p/>
 * <execute-fitnessse-tests
 * suitepage="FitNesse.SuiteAcceptanceTests"
 * fitnessreport="8082"
 * resultsdir="${results.dir}"
 * resultshtmlpage="fit-results.html"
 * classpathref="classpath" />
 * </pre>
 */
```

## Nelokální informace

Jestliže musíte napsat komentář, ujistěte se, že popisuje kód, který je poblíž. Neposkytujte v kontextu lokálního komentáře informace týkající se celého systému. Podívejte se například na níže uvedený komentář pro javadoc. Kromě skutečnosti, že je příšerně redundantní, nabízí také informaci ohledně implicitního portu. A přesto nemá funkce naprostě žádnou kontrolu nad tím, co jaká ta implicitní hodnota je. Poznámka nepopisuje funkci, ale nějakoujinou, velmi vzdálenou část systému. Samozřejmě neexistuje záruka, že tento komentář bude změněn, když bude změněn kód, obsahující implicitní hodnotu.

```
/**  
 * Port, na kterém by běžel test. Implicitně <b>8082</b>.   
 *  
 * @param fitnessPort  
 */  
public void setFitnessPort(int fitnessPort)  
{  
    this.fitnessPort = fitnessPort;  
}
```

## Příliš mnoho informací

Nevkládejte do svých komentářů zajímavé historické diskuse nebo nedůležité popisy detailů. Níže uvedený komentář byl vybrán z modulu, který byl určen k testování, zda může funkce zakódrovat a dekódovat formát ve formátu base64. Žádný čtenář tohoto kódu nemá žádnou potřebu zkoumat tajemné informace obsažené v tomto komentáři, kromě pořadového čísla RFC (request for comments).

```
/*  
RFC 2045 - Multipurpose Internet Mail Extensions (MIME)  
Část první: Formátování těl internetových zpráv, sekce 6.8.  
Base64 Content-Transfer-Encoding  
Kódovací proces reprezentuje skupiny po 24 vstupních bitech jako  
výstupní řetězce se čtyřmi zakódovanými znaky. Při zpracování zleva  
doprava se vytváří 24bitová skupina zřetězením tří 8bitových vstupních  
skupin. 24 bitů se pak zpracovává jako 4 zřetězené 6bitové skupiny,  
z nichž je každá převedena na jednu číslici abecedy zakódované v Base64.  
Během kódování bitového proudu pomocí Base 64 předpokládáme, že bitový  
proud je uspořádán tak, že nejvýznamnější bit je první.  
To znamená, že první bit prvního 8bitového znaku v datovém proudu bude  
mít nejvyšší řád, a osmý bit tohoto znaku bude mít nejnižší řád, atd.  
*/
```

## Nejasná spojitost

Spojitost mezi komentářem a kódem, který popisuje, by měla být jasná. Chcete-li způsobit nepříjemnost a napsat nějaký komentář, pak nechť si čtenář může alespoň přečíst komentář i kód a porozumět tomu, o čem komentář mluví. Podívejte se například na tento komentář opsaný z komponent apache commons (a přeložený):

```
/*
 * začněte s dostatečně velkým polem, aby do něj bylo možné uložit
 * všechny pixely (plus bajty filtru), a dalších 200 byteů pro informaci ze záhlaví
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Co znamenají bajty filtru? Vztahuje se k operaci +1? Nebo k operaci \*3? Je pixel bajt? Proč 200? Záměr komentáře je vysvětlovat kód v těch případech, kdy se kód není schopen vysvětlit sám. Je škoda, když komentář potřebuje další vysvětlení.

## Záhlaví funkcí

Krátké funkce nepotřebují příliš mnoho popisování. Dobře zvolený název malé funkce, která provádí jen jeden úkol, je obvykle lepší než záhlaví s komentáři.

## Javadoc v neveřejném kódu

Nakolik jsou komentáře pro javadoc užitečné u veřejných API, natolik jsou prokletím u kódu, který není určen pro veřejnost. Generování stránek pomocí javadoc pro třídy a funkce uvnitř systému není obecně užitečné, a zvláštní formalismus komentářů javadoc nepřispívá k ničemu jinému, než k redundancím a odvádění pozornosti.

## Příklad

Modul ve výpisu 4.7 jsem napsal pro první kurs XP. Měl to být příklad špatného kódu a špatného způsobu komentování. Poté tento kód refaktoroval Kent Black do mnohem příjemnější formy před zraky několika desítek nadšených studentů. Později jsem tento příklad adaptoval pro svou knihu *Agile Software Development, Principles, Patterns, and Practices* a pro první z mých článků *Craftsman*, publikovaných v časopise *Software Development*.

Na tomto modulu je fascinující, že mnozí z nás jej byli schopni někdy označit jako „dobře dokumentovaný“. Nyní v něm vidíme určitý nepořádek. Podívejte se, kolik tam najdete různých a problematických komentářů.

### Výpis 4.7. GeneratePrimes.java

```
/**
 * Tato třída generuje prvočísla až do maximální velikosti, kterou
 * stanoví uživatel. Použitý algoritmus je Eratosthenovo síto.
 * <p>
 * Eratosthenes z Kyrény, nar. 276 př.n.l., Lybie
 * zemřel r. 194 př.n.l. v Alexandrii. Jako první člověk spočítal
 * obvod Země. Rovněž je známo, že se zabýval kalendářem s přestupnými
 * roky a vedl Alexandrijskou knihovnu.
 * <p>
 * Algoritmus je docela jednoduchý. Je dáno pole celočíselných hodnot
 * s počáteční hodnotou 2. Vyškrtněte všechny násobky dvou. Najděte
 * následující nezaškrtnuté číslo a vyškrtněte všechny jeho násobky.
 * Postup opakujte tak dlouho, dokud nepřekročíte druhou odmocninu
 * z maximální hodnoty.
 */
```

```
* @author Alphonse
* @version 13 Feb 2002 atp
*/
import java.util.*;
public class GeneratePrimes
{
    /**
     * @param maxValue je omezení pro generování čísel.
     */
    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue >= 2) // jediný platný případ
        {
            // deklarace
            int s = maxValue + 1; // velikost pole
            boolean[] f = new boolean[s];
            int i;
            // inicializuj pole na hodnotu true.
            for (i = 0; i < s; i++)
                f[i] = true;
            // zbať se známých neprvočísel
            f[0] = f[1] = false;
            // síto
            int j;
            for (i = 2; i < Math.sqrt(s) + 1; i++)
            {
                if (f[i]) // je-li i nezaškrtnuté, vyškrtni jeho násobky.
                {
                    for (j = 2 * i; j < s; j += i)
                        f[j] = false; // násobek není prvočíslem
                }
            }
            // kolik je tam prvočísel?
            int count = 0;
            for (i = 0; i < s; i++)
            {
                if (f[i])
                    count++; // čítač přeskočených čísel.
            }
            int[] primes = new int[count];
            // přesuň prvočísla do výsledků
            for (i = 0, j = 0; i < s; i++)
            {
                if (f[i]) // jde-li o prvočíslo
                    primes[j++] = i;
            }
            return primes; // vrát hodnotu prvočísla
        }
        else // maxValue < 2
            return new int[0]; // jsou-li vstupní data chybná, vrát pole s hodnotami null.
    }
}
```

Ve výpisu 4.8 uvidíte verzi toho modulu po refaktorování. Všimněte si, že používání komentářů je významně omezeno. V celém modulu jsou jen dva komentáře. Oba jsou svou podstatou vysvětlující.

**Výpis 4.8.** GeneratePrimes.java (po refaktorování)

```
/**  
 * Tato třída generuje prvočísla až do maxima stanoveného  
 * uživatelem. Použitý algoritmus je Eratosthenovo síto.  
 * Je dáno pole typu integer, začínající hodnotou 2:  
 * Nalezněte první nepřeškrtnuté celé číslo a vyškrtněte  
 * všechny jeho násobky. Postup opakujte, dokud jsou v poli  
 * nějaké násobky.  
 */  
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()  
    {  
        // Každý násobek v poli má prvočinitele, který  
        // je menší nebo roven odmocnině z délky pole,  
        // takže nemusíme vyškrvat násobky čísel  
        // větších než je tato odmocnina.  
        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 number_of_UncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            count++;
    return count;
}
```

Je jednoduché argumentovat, že první komentář je zbytečný, protože jej čteme jako vlastní název funkce `generatePrimes`. Přesto si myslím, že tento komentář zjednodušuje čtenáři chápání algoritmu a spíše bych ho v kódu ponechal.

Druhý komentář je určitě nezbytný. Vysvětluje důvody, proč byla použita druhá odmocnina jako mez cyklu. Nedokázal jsem najít ani jednoduchý název proměnné, ani nějakou jinou strukturu kódování, která by tento bod vyjasnila. Na druhé straně by použití druhé odmocniny mohl být dobrý nápad. Opravdu šetřím tolik času, když omezím iteraci druhou odmocninou? Mohl by výpočet druhé odmocniny spotřebovat víc času, než kolik ho ušetřím?

Stojí to za úvahu. Použití druhé odmocniny jako horní meze cyklu mě uspokojuje coby céčkového a asemblerového hackera, ale nejsem přesvědčen, že to ostatním stojí za ten čas a úsilí tomu porozumět.

## Použitá literatura

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



# KAPITOLA 5

## Formátování

**V této kapitole najdete:**

- ◆ Důvody formátování
- ◆ Vertikální formátování
- ◆ Horizontální formátování
- ◆ Týmová pravidla
- ◆ Formátovací pravidla strýčka Boba



Když se lidé podívají pod kapotu, chceme, aby to na ně udělalo dojem a aby jim imponovala úhlednost, konzistence a pozornost věnovaná detailu. Chceme, aby je udeřila do očí uspořádanost. Chceme, aby pozvedli obočí, když budou procházet jednotlivými moduly. Chceme, aby pochopili, že tady pracovali profesionálové. Když místo toho uvidí směs kódů, který vypadá, jakoby byl napsán skupinou opilých námořníků, budou mít tendenci udělat závěr, že stejná nedbalost k detailům bude převládat i ve všech ostatních stránkách projektu.

Měli byste se postarat o to, aby byl váš kód pěkně formátovaný. Měli byste si vybrat sadu jednoduchých pravidel, kterými se bude formátování vašeho kódu řídit, a tato pravidla byste měli konzistentně používat. Pokud pracujete v týmu, měl by celý tým souhlasit s jedinou sadou pravidel formátování a všichni členové by se jí měli řídit. Je dobré mít automatický nástroj, s jehož pomocí je možné tato pravidla aplikovat.

## Důvody formátování

Především mluvme jasně. Formátování kódu je *důležité*. Je příliš důležité na to, abychom je ignorovali nebo abychom je brali nábožensky. Formátování kódu je záležitostí komunikace a komunikace je u profesionálního vývojáře prvořadým úkolem.

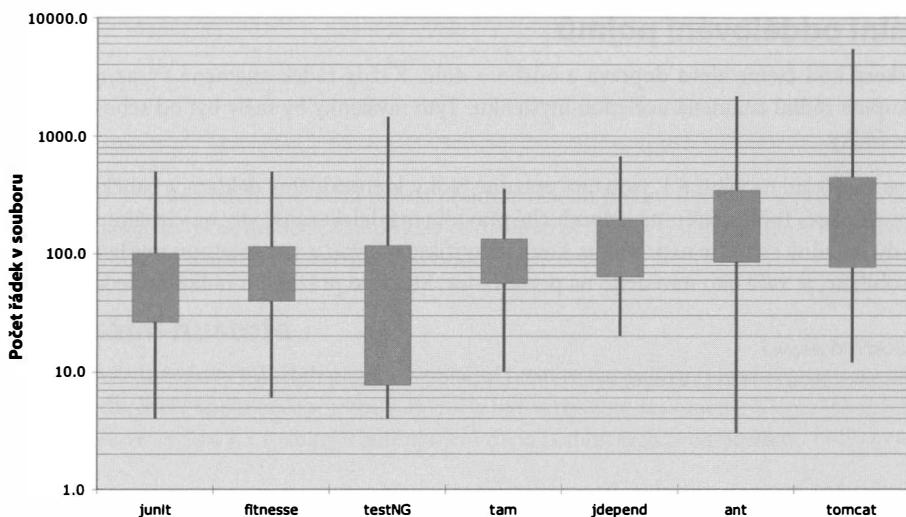
Možná si myslíte, že prvořadým úkolem profesionálního vývojáře je „aby to fungovalo“. Nyní ale doufám, že vás tato kniha tohoto omylu zbabila. Je pravděpodobné, že funkcionality, kterou vytváříte dnes, bude v příští verzi změněna, ale čitelnost vašeho kódu bude mít zásadní vliv na všechny změny, které kdy budete provádět. Styl kódování a čitelnost vytváří precedens, který bude i nadále ovlivňovat udržovatelnost a rozšířitelnost ještě dlouho poté, co se původní kód změní k nepoznání. Váš styl a disciplína přežije, i když kód nikoliv. Takže co skrývají otázky formátování, které nám pomůže komunikovat co nejlépe?

## Vertikální formátování

Začneme s vertikálním rozsahem. Jak by měl být zdrojový kód dlouhý? V Javě je délka souboru těsně spjata s velikostí třídy. Když mluvíme o třídách, mluvíme o velikosti třídy. Zkusme se v tento okařík zamyslet jen nad velikostí souboru.

Jak velká je většina zdrojových souborů v Javě? Vypadá to tak, že existuje značný rozptyl velikostí a jsou některé pozoruhodné rozdíly ve stylu. Obrázek 5.1 ukazuje některé z těchto rozdílů. Zobrazuje sedm různých projektů: JUnit, FitNesse, testNG, Time and Money, JDepend, Ant a Tomcat. Řádky procházející obdélníky ukazují minimální a maximální délku každého z projektů. Obdélníky zobrazují přibližně jednu třetinu (standardní odchylku<sup>1</sup>) souborů. Střed obdélníku je průměr. Takže průměrná velikost souboru v projektu FitNesse je kolem 65 řádek a zhruba jedna třetina souborů je mezi 40 a něco málo nad 100 řádků. Největší soubor ve FitNesse má přibližně 400 řádků a nejménší jich má 6. Všimněte si, že se jedná o logaritmické měřítko, takže malý rozdíl ve vertikální pozici znamená velký rozdíl v absolutní velikosti.

1. Obdélník znázorňuje  $\sigma/\sqrt{2}$  nad a pod průměrem. Ano, vím, že distribuce délek souborů nepodléhá normálnímu rozložení a stejně tak standardní odchylka není matematicky přesná. Ale zde nám nejdou o přesnost. Pouze se snažíme o první seznámení.



Obrázek 5.1. Rozdělení délek souborů v logaritmickém měřítku (výška obdélníku = sigma)

Junit, FitNesse a Time and Money se skládají z relativně malých souborů. Žádný z nich nemá více než 500 řádků a většina z těchto souborů má méně než 200 řádků. Na druhé straně Tomcat a Ant mají několik souborů, které mají několik tisíc řádků, a téměř polovina z nich má přes 200 řádků.

Co to pro nás znamená? Vypadá to tak, že je možné vytvořit významné systémy (FitNesse má délku téměř 50 000 řádků) ze souborů, jejichž typická délka je 200 řádků s hornímezí 500 řádků. Ačkoliv by tohle nemělo být neméně pravidlo, měli bychom je považovat za velmi žádoucí. Malé soubory jsou obvykle srozumitelnější než soubory velké.

## Přirovnání k novinám

Podívejte se na dobře napsaný novinový článek. Ten čtete vertikálně. Na začátku očekáváte titulek, který vám řekne, o čem bude pojednávat, a umožní vám rozhodnout se, zda se jedná o něco, co chcete číst. První odstavec vám poskytne přehled celé zprávy, nepojednává o detailech a nastíní vám její hrubé rysy. Jak pokračujete dále, počet detailů roste a dovidáte se veškerá data, názvy, citáty, tvrzení a ostatní podrobnosti.

Zdrojový soubor by se měl podobat novinovému článku. Název by měl být jednoduchý, ale vysvětlující. Samotný název by měl stačit k tomu, aby nám sdělil, zda jsme ve správném modulu či nikoliv. Horní části zdrojového souboru by měly poskytnout pojmy a algoritmy vysoké úrovni. Výčet detailů by se měl zvyšovat s tím, jak se pohybujeme dolů, a na konci zdrojového souboru bychom měli najít funkce a detaily nejnižší úrovni.

Noviny obsahují mnoho článků. Většina z nich je krátká. Některé jsou o něco delší. Jen málo z nich je na celou stránku. To dělá z novin něco použitelného. Kdyby byla v novinách jen jedna dlouhá zpráva obsahující dezorganizované seskupení faktů, dat a názvů, prostě bychom je nečetli.

## Vertikální oddělování pojmu

Téměř veškerý kód čteme zleva doprava a odshora dolů. Každý řádek znamená výraz nebo větu a každá skupina řádků znamená ucelenou myšlenku. Tyto myšlenky by měly být od sebe odděleny prázdnými řádky.

Podívejte se například na výpis 5.1. Jsou tam prázdné řádky, které oddělují deklarace balíčku, importu a všechny funkce. Tohle extrémně jednoduché pravidlo má dalekosáhlý vliv na vizuální strukturu kódu. Každý prázdný řádek je nápovedou, která identifikuje novou a samostatnou myšlenku. Když si výpis prohlížíte, je vaše oko navedeno na první řádek, který po prázdném řádku následuje.

### Výpis 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();
    }
}
```

Když odstraníte prázdné řádky, jako na výpisu 5.2, čitelnost kódu se tím pozoruhodně zamlží.

### Výpis 5.2. BoldWidgdet.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();  
    }  
}
```

Tento vliv je ještě zřetelnější, když očima přejdete někam jinam. Rozdílné seskupení řádků v prvním příkladě vás trkne do očí, zatímco kód ve druhém případě vypadá zmateně. Rozdíl mezi těmito dvěma výpisy je částečně i ve vertikálním oddělování.

## Vertikální hustota

Jestliže prázdné řádky oddělují pojmy, vertikální hustota naopak znamená těsnou souvislost. Takže řádky kódu, které spolu úzce souvisejí, by měly být vertikálně seskupeny. Všimněte si, jak zbytečné komentáře ve výpisu 5.3 rozvíjejí těsné spojení dvou instančních proměnných.

### Výpis 5.3. Zbytečné komentáře snižující přehlednost

```
public class ReporterConfig {  
    /**  
     * Jméno třídy přijímače reportéra  
     */  
    private String m className;  
    /**  
     * Vlastnosti přijímače reportéra  
     */  
    private List<Property> m properties = new ArrayList<Property>();  
    public void addProperty(Property property) {  
        m properties.add(property);  
    }
```

Výpis 5.4 je mnohem čitelnější. Umí „padnout do oka“, nebo alespoň v mé případě. Mohu si jej prohlížet a vidět zde jednu třídu se dvěma proměnnými a jednou metodou, aniž bych musel příliš zrakem přelétat z místa na místo. Předchozí výpis mě příliš k témtoto úkonům nutí jen proto, abych nakonec dospěl k tomu samému.

### Výpis 5.4. Vynechání zbytečných komentářů

```
public class ReporterConfig {  
    private String m className;  
    private List<Property> m properties = new ArrayList<Property>();  
    public void addProperty(Property property) {  
        m properties.add(property);  
    }
```

## Vertikální vzdálenost

Už jste se někdy lopotili třídou a skákali z jedné funkce na druhou, rolovali zdrojovým kódem a zkoušeli uhádnout, jak pracují funkce a jaké jsou mezi nimi vazby, jen abyste uvízli v pasti chaosu a zmatku? Už jste někdy pátrali v dědické hierarchii po definici proměnné nebo funkce? Je to frustrující,

protože se pokoušíte zjistit, co systém dělá, ale trávíte svůj čas a duševní energii, abyste zjistili, kde se jednotlivé díly nacházejí.

Myšlenky, které spolu těsně souvisejí, by měly být ve svislém směru blízko sebe [O10]. Tohle pravidlo zřejmě nefunguje pro ty pojmy, které patří do samostatných souborů. Ale pak by se úzce související pojmy neměly do samostatných souborů umisťovat, pokud pro to nemáte opravdu dobré důvody. Tohle je opravdu jeden z důvodů, proč se vyhýbat používání chráněných proměnných. Pro pojmy, které jsou tak těsně spjaty, že náležejí do stejného zdrojového souboru, by jejich vertikální oddělení mělo být mírou, nakolik je každý z nich důležitý pro pochopení toho druhého. Chceme se vyhnout tomu, abychom nutili naše čtenáře přecházet z jedné třídy nebo jednoho zdrojového souboru do druhého.

**Deklarace proměnných.** Proměnné by měly být deklarovány co nejbližše místa, kde se používají. Protože jsou naše funkce velmi krátké, lokální proměnné by se měly objevit na začátku každé funkce, jak tomu je u této poněkud delší funkce z JUnit4.3.1.

```
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) {
        }
    }
}
```

Řídicí proměnné cyklů by měly být obvykle deklarovány v rámci tohoto příkazu, jako v této chytré malé funkci ze stejného zdroje:

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

V některých řídkých případech u delších funkcí lze deklarovat proměnnou na začátku bloku nebo těsně před cyklem. Takovou proměnnou můžete vidět v části kódu zevnitř velmi dlouhé funkce 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);
    }
}
...
}
```

**Instanční proměnné** by na druhé straně měly být deklarovány na počátku třídy. To by nemělo zvyšovat vertikální vzdálenost těchto proměnných, protože v dobré navržené třídě je používá mnoho metod, pokud ne všechny.

Na téma, kam by se měly umístit instanční proměnné, již proběhlo mnoho debat. V jazyce C++ používáme obyčejně *pravidlo nůžek*, podle kterého se všechny instanční proměnné umístí na konec. Avšak obecná konvence v jazyce Java je umístit je na začátek třídy. Nevidím důvod, proč se řídit kteroukoliv jinou konvencí. Pro instanční proměnné je důležité, aby byly deklarovány na jednom dobré známém místě. Každý by měl vědět, kde se na ně může podívat.

Podívejte se například na zvláštní případ třídy `TestSuite` v JUnit 4.3.1. Tuto třídu jsem hodně zredukoval, abych se dostal k jádru věci. Jestliže se podíváte zhruba doprostřed výpisu, uvidíte zde dvě deklarace instančních proměnných. Těžko by je někdo mohl ukryt na lepší místo. Kdokoliv, kdo bude tento kód číst, o ně bude muset náhodou zakopnout, jak se to stalo mně.

```
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) {
        ...
    }
    ...
}
```

**Závislé funkce.** Jestliže jedna funkce volá druhou, měly by být vertikálně blízko sebe, a volající funkce by měla být nad funkcí volanou, pokud to je možné. To dává programu přirozený běh. Pokud se toto pravidlo spolehlivě dodržuje, čtenáři mohou věřit, že definice funkcí budou následovat krátce po jejich použití. Podívejte se například na kousek kódu z FitNesse ve výpisu 5.5. Všimněte si, jak nejvíše umístěné funkce volají ty pod nimi a ty opět volají jiné, umístěné níže. To usnadňuje hledání volaných funkcí a značně zlepšuje čitelnost celého modulu.

#### Výpis 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;
    }
    ...
}
```

Jen na okraj, tento kousek kódu nám dává pěkný příklad umístění konstant na odpovídající úrovni [O35]. Konstanta "FrontPage" by mohla být ukryta ve funkci `getPageNameOrDefault`, ale to by skrylo dobře známou a předpokládanou konstantu ve funkci nízké úrovně bez zřejmého důvodu. Je lepší předat ji směrem dolů z místa, kde má smysl ji znát, na místo, kde se skutečně používá.

**Konceptuální afinita.** Některé části kódu *chtějí* být blízko jiných částí. Mají určitou konceptuální – pojmovou – souvislost. Čím je silnější, tím by mezi nimi měla být vertikální vzdálenost menší.

Jak jsme již viděli, tato afinita by mohla být založena na přímé závislosti, jako je volání jedné funkce jinou nebo jako je použití proměnné funkcí. Ale afinita může být způsobena tím, že skupina funkcí provádí podobné operace. Podívejte se na část kódu 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);
    }
    ...
}
```



Tyto funkce mají silnou konceptuální afinitu, protože sdílejí společné schéma tvorby jmen a provádějí různé varianty stejného základního úkolu. Skutečnost, že jedna funkce volá druhou, je druhotná. I kdyby nevolaly, stejně by měly být blízko sebe.

## Vertikální uspořádání

Obecně chceme, aby závislosti volání funkcí směřovaly dolů. To znamená, že volaná funkce by měla být pod funkcí, která ji volá.<sup>2</sup>

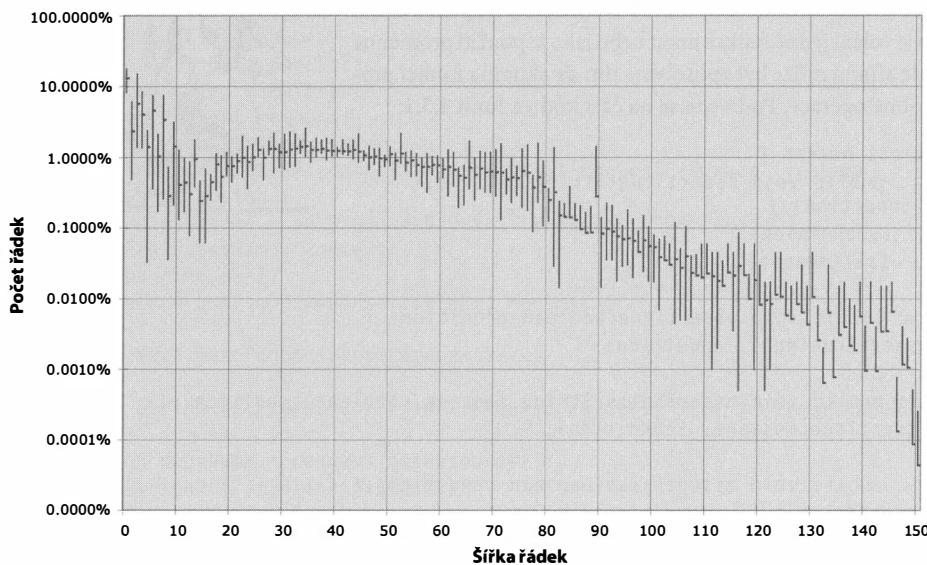
To vytváří v modulu úhledný tok zdrojového kódu směrem dolů od vyšší úrovně k nižší.

Podobně jako u novinových článků předpokládáme, že nejdůležitější myšlenky budou na prvním místě a že budou vyjádřeny s minimálním množstvím zatemňujících detailů. Předpokládáme, že detaily nízké úrovně budou na posledním místě. To nám umožní, abychom ze zdrojových souborů sbírali smetanu a dostali podstatu z prvních několika funkcí bez nutnosti se vnořovat do detailů. Takto je organizován výpis 5.5. Možná, že lepší příklady najeznete ve výpisech 15.5 na straně 270 a 3.7 na straně 70.

2. Je to pravý opak toho, co platí pro jazyky Pascal, C a C++, kde je nutné funkce definovat, nebo alespoň deklarovat *před* jejich použitím.

## Horizontální formátování

Jak by měl být rádek široký? Abychom zodpověděli tuto otázku, podívejme se na to, jak jsou široké řádky u typických programů. Obrázek 5.2 ukazuje rozdělení délek řádků u všech sedmi projektů. Pravidelnost je impozantní, zvláště těsně kolem 45 znaků. Skutečně, každá jednotka v rozmezí mezi 20 až 60 reprezentuje kolem jednoho procenta celkového počtu řádků. To je 40 procent! Možná, že dalších 30 procent z nich jsou kratší, než je 10 řádků. Nezapomeňte, že jde o logaritmické měřítko, takže prudký pokles nad 80 znaků, který vypadá lineárně, je skutečně velmi prudký. Programátoři určitě preferují krátké řádky.



Obrázek 5.2. Rozdělení šířky řádek v jazyce Java

To by mohlo napovídat, že bychom se měli snažit o krátké řádky. Staré omezení formátu Hollerith, které bylo 80 znaků, je nepovinné a já bych neměl nic proti řádkům s délkou až 100 nebo 120 znaků. Ale za touto hranicí jde spíše o lehkomyslnost.

Kdysi jsem se řídil pravidlem, abychom nemuseli nikdy rolovat směrem doprava. Ale monitory jsou dnes skutečně široké a mladší programátoři mohou zmenšit font natolik, aby se jim na obrazovku vešlo 200 znaků. To nedělejte. Já si osobně nastavuji limit na 120 znaků.

## Horizontální oddělování a hustota

Horizontálně používáme mezery ke spojení toho, co má těsnou souvislost, a k oddělení toho, co spolu souvisí slabě. Podívejte se na následující funkci:

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

Obklopil jsem přiřazovací operátory mezerami, abych je zvýraznil. Přiřazovací příkazy mají dva zřetelné a důležité prvky: levou a pravou stranu. Mezery totiž oddělují zviditelnější.

Na druhé straně jsem neumístil mezery mezi názvy funkcí a počáteční závorky. Je to proto, že funkce a její argumenty spolu těsně souvisejí. Jejich oddělení by ukazovalo, že spolu nesouvisejí. Ve volání funkce v jejich závorkách argumenty odděluji, abych zvýraznil čárku a ukázal, že argumenty jsou samostatné.

Jiné použití mezer spočívá ve zvýraznění priority operátorů.

```
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;
    }
}
```

Všimněte si, jak dobře se rovnice čtou. Zlomky nemají mezi sebou žádné mezery, protože mají vysokou prioritu. Jednotlivé členy jsou odděleny mezerami, protože sčítání a odečítání mají nižší prioritu.

Bohužel většina nástrojů pro reformátování kódu nepřihlíží k prioritám operátorů a všude vkládají mezery stejným způsobem. Takže se pečlivé vkládání mezer, jak je uvedeno výše, po reformátování kódu zpravidla ztratí.

## Horizontální zarovnání

Když jsem byl programátorem v asembleru<sup>3</sup>, používal jsem horizontální zarovnání, abych zvýraznil určité struktury. Když jsem začal programovat v jazyce C, C++ a eventuálně v Javě, pokračoval jsem v praxi zarovnávat všechny názvy proměnných v řadě deklarací nebo všechny pravé strany v řadě přiřazovacích příkazů. Můj kód mohl vypadat takto:

```
public class FitNesseExpeditor 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;
```

3. Co to říkám? Stále jsem programátorem v asembleru. Klukovi hračku nevezmete.

```

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;
}

```

Zjistil jsem však, že tento druh zarovnání není vhodný. Tohle zarovnání zřejmě zvýrazňuje nesprávné rysy a odvádí mě od skutečného záměru. Například ve výše uvedeném seznamu deklarací jste v pokušení číst seznam jmen proměnných odshora dolů, aniž byste se dívali na jejich typy. Podobně v seznamu přiřazovacích příkazů jste v pokušení prohlížet si r-hodnoty, aniž byste se kdy podívali na přiřazovací operátor. Co je horší, nástroje pro automatické reformátování obvykle tento druh zarovnání odstraní.

Takže výsledkem je, že tento druh zarovnání už nepoužívám. Nyní dávám přednost nezarovnaným deklaracím a přiřazovacím příkazům, jak uvádím níže, protože zdůrazňují zásadní vadu. Pokud mám dlouhý seznam, který potřebuji zarovnat, *problémem je délka seznamu* a ne to, že nepoužiji zarovnání. Délka seznamu deklarací v následující třídě `FitNesseExpediter` naznačuje, že by se třída měla rozdělit.

```

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;
    }

```

## Odsazování

Zdrojový soubor je spíše hierarchií, trochu jako osnova. Jsou tam informace, které se vztahují k celému souboru, k jednotlivým třídám v souboru, k metodám tříd, k blokům v rámci metod a rekurzivně k jejich podřízeným blokům. Každá úroveň této hierarchie je oblastí, ve které je možné deklarovat názvy, v nichž jsou interpretovány deklarace a proveditelné příkazy.

Abychom tuto hierarchii oblastí zviditelnili, v rámci této hierarchie odsazujeme řádky zdrojového kódu úměrně jejich pozici. Příkazy na úrovni souboru, jako většina deklarací tříd, nejsou odsazeny vůbec. Metody v rámci třídy jsou odsazeny jednu úroveň vpravo od deklarace metody. Implementace podřízených bloků jsou realizovány jednu úroveň vpravo od svého nadřízeného bloku atd.

Programátoři se na tohle schéma odsazování dost spoléhají. Vizuálně řadí jednotlivé řádky vlevo tak, aby viděli, ve kterém rozsahu se nachází. To jim umožňuje tyto oblasti rychle přeskočit, jako třeba implementace příkazů `if` nebo `while`, které nejsou v dané situaci relevantní. Mohou prolétnout levou část a hledat nové deklarace metod, nové proměnné a dokonce i nové třídy. Bez odsazování by byly programy pro lidi prakticky nečitelné. Podívejte se na následující programy, které jsou syntakticky a sémanticky identické:

```
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();
    }
}
}
```

Vaše oko rychle rozpozná strukturu odsazeného souboru. Téměř okamžitě uvidíte proměnné, konstruktory, objekty přístupu a metody. Zabere vám to sotva pár sekund, než zjistíte, že tohle je nějaké jednoduché zapouzdření soketu s časovým limitem. Verze bez odsazení je bez intenzivního studia prakticky nesrozumitelná.

**Nedodržování pravidel odsazování.** Někdy je lákavé pravidla odsazování porušit, třeba u krátkých příkazů `if`, cyklů `while` nebo krátkých funkcí. Kdykoliv jsem tomuto pokušení podlehl, téměř vždy

jsme se musel vrátit zpět a uvést odsazení do původního stavu. Takže se vyhýbám redukování rozsahu těla příkazu na jeden řádek, jak je to zde:

```
public class CommentWidget extends TextWidget
{
    public static final String REGEXP = "^#[^\r\n]*(?:\r\n|\n|\\r)?";
    public CommentWidget(Widget parent, String text){super(parent, text);}
    public String render() throws Exception {return "";}
}
```

Raději příkazy rozvinu a odsadím, jako zde:

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

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

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

## Prázdné obory

Někdy je tělo příkazu `while` nebo `for` prázdné, jak vidíte níže. Nemám tyto struktury rád a snažím se jim vyhnout. Když to není možné, postarám se o to, aby prázdné tělo bylo náležitě odsazeno a uzavřeno závorkami. Nevím, kolikrát jsem se nechal zmást středníkem, který si těše hověl na konci řádku cyklu `while`. Pokud středník na jeho vlastním řádku nezviditelníte náležitým odsazením, není jednoduché si jej všimnout.

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

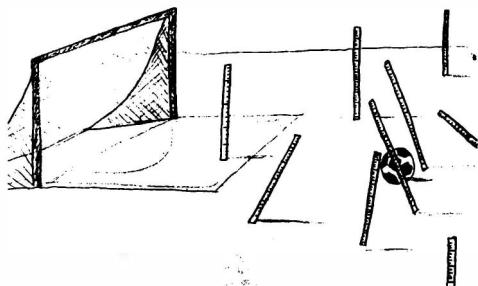
## Týmová pravidla

Každý programátor má svá oblíbená formátovací pravidla, ale pokud pracuje v týmu, pak je to tým, kdo určuje pravidla.

Tým vývojářů by se měl shodnout na jednom stylu formátování a každý člen týmu by jej pak měl používat. Chceme, aby software měl konzistentní styl. Nechceme, aby se zdálo, že jej psala parta rozhádaných individuí.

Když jsem začal v roce 2002 pracovat na projektu

FitNesse, sedli jsme si společně s týmem, abychom vypracovali styl psaní kódu. To nám zabralo asi 10 minut. Dohodli jsme se, kam bychom měli umísťovat závorky, jak velké bude naše odsazení, jak



budeme pojmenovávat třídy, proměnné a metody atd. Pak jsme tato pravidla zakódovali v našem IDE do nástroje pro formátování kódu a od té doby jsme tato pravidla neustále dodržovali. Nebyla to pravidla, která bych upřednostňoval, ale byla to pravidla stanovená týmem. Jako jeho člen jsem je během psaní kódu pro projekt FitNesse dodržoval.

Nezapomínejte, že dobrý softwarový systém se skládá z mnoha dokumentů, které se dobře čtou. Je potřeba, aby měly konzistentní a bezproblémový styl. Čtenář potřebuje věřit, že informace ukrytá ve formátování, kterou uvidí v jednom zdrojovém souboru, bude znamenat totéž i v těch ostatních. Poslední věcí, kterou bychom chtěli udělat, je zkomplikovat zdrojový kód tím, že jej napíšeme mnoha různými individuálními styly.

## Formátovací pravidla strýčka Boba

Pravidla, která používám, jsou velmi jednoduchá a jsou zobrazena ve výpisu kódu 5.6. Podívejte se na tento příklad, kde kód sám představuje nejlepší popis standardu kódování.

### Výpis 5.6. CodeAnalyzer.java

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxWidth;
    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));
        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("Sem by se program neměl dostat");
    }
    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;
    }
}
```

# KAPITOLA 6

## Objekty a datové struktury

**V této kapitole najdete:**

- ◆ Datové abstrakce
- ◆ Datová a objektová antisymetrie
- ◆ Démétřin zákon
- ◆ Objekty pro přenos dat
- ◆ Shrnutí
- ◆ Literatura



Existují důvody, proč používat privátní proměnné. Nechceme, aby na nich závisel někdo jiný. Chceme mít možnost měnit jejich typ nebo implementaci podle naší libovůle nebo z jiných podnětů. Proč tedy také programátorů přidává automaticky do svých objektů přístupové metody a zveřejňuje své privátní proměnné, jako kdyby byly veřejné?

## Datové abstrakce

Podívejte se na rozdíl mezi výpisy 6.1 a 6.2. Oba představují kartézské souřadnice bodu v rovině. A přesto jeden z nich zveřejňuje jeho implementaci a druhý ji zcela skrývá.

### Výpis 6.1. Konkrétní bod

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

### Výpis 6.2. Abstraktní bod

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

Na výpisu 6.2 je jedna pěkná věc. Neexistuje způsob, jak určit, zda se jedná o implementaci kartézských nebo polárních souřadnic. Nemusí to být ani jedna. A přesto rozhraní nepochybně reprezentuje datovou strukturu.

Ale reprezentuje to něco více než jen datovou strukturu. Metoda určuje zásady přístupu. Můžete číst jednotlivé souřadnice nezávisle, ale jejich nastavení musí proběhnout najednou jako nedělitelná operace.

Na druhé straně výpis 6.1 ukazuje jasnou implementaci kartézských souřadnic a nutí nás pracovat se souřadnicemi nezávisle. To jejich implementaci zveřejňuje vnějšímu okolí. Skutečně, jejich implementace by byla zveřejněna, i kdyby byly proměnné soukromé a my bychom používali přístupové metody pro jednoduché proměnné.

Skrývání implementace není jen otázkou vkládání vrstvy funkcí mezi proměnné. Skrývání implementace je otázkou abstrakcí! Nejde o to, že třída zpřístupňuje své proměnné pomocí přístupových metod. Jde o to, že zveřejní abstraktní rozhraní, které uživatelům umožní pracovat s *podstatou* dat, aniž by museli znát jejich implementaci.

Podívejte se na výpisy 6.3 a 6.4. První z nich používá pro sdělení hladiny paliva v autě konkrétní termíny, zatímco druhý to dělá pomocí abstrakce procent. V konkrétním případě si můžete být opravdu jisti, že jde pouze o přístupové objekty k proměnným. V abstraktním případě nemáte ani ponětí o tom, v jaké formě data jsou.

**Výpis 6.3.** Konkrétní auto

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

**Výpis 6.4.** Abstraktní auto

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

Z obou výše uvedených případů je přijatelnější ten druhý. Nechceme zveřejňovat detaily svých dat. Raději je ukážeme v abstraktní formě. Toho nedosáhneme pouhým použitím rozhraní nebo přístupových metod. Nad nejlepším způsobem, jak zobrazit data objektu, je třeba se vážně zamyslet. Nejhorší volbou je slepé přidávání přístupových metod.

## Datová a objektová antisymetrie

Tyto dva příklady ukazují rozdíl mezi objekty a datovými strukturami. Objekty skrývají svá data za abstrakce a zveřejňují funkce, které s nimi pracují. Datové struktury zveřejňují svá data a nemají žádné užitečné funkce. Vraťte se a přečtěte si to ještě jednou. Všimněte si komplementární povahy obou definic. Prakticky to jsou protiklady. Tento rozdíl vám může připadat triviální, ale má dalekosáhlé důsledky.

Podívejte se například na výpis 6.5, který obsahuje kód zpracovávající tvary. Třída Geometry pracuje se třemi třídami tvarů. Tyto třídy jsou jednoduché datové struktury bez jakékoliv funkčnosti. Veškerá funkčnost je ve třídě Geometry.

**Výpis 6.5.** Procedurální zpracování tvarů

```
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();
    }
}

```

Objektově orientovaní programátoři by nad tím mohli ohrnovat nosy a namítat, že jde o procedurální kód – a měli by pravdu. Ale jejich pošklebování by nemuselo být oprávněné. Zvažte, co by se stalo, kdybychom do třídy `Geometry` přidali funkci `perimeter()`. Třídy pro zpracování tvarů by se nezměnily! Jakákoli jiná třída, která je nich závislá, by se také nezměnila! Na druhé straně, když přidám třídu pro nový tvar, musím změnit všechny funkce ve třídě `Geometry`, aby ho správně zpracovaly. Ještě jednou si to přečtěte. Všimněte si, že tyto dvě podmínky jsou diametrálně odlišné.

Nyní se podívejte na objektově orientované řešení na výpisu 6.6. Zde máme metodu `area()`, která je polymorfní. Nepotřebujeme žádnou třídu `Geometry`. Takže když přidám nový tvar, není tím ovlivněna žádná z existujících funkcí, ale když přidám novou funkci, musím změnit všechny třídy *tvarů*!<sup>11</sup>

#### Výpis 6.6. Polymorfní zpracování tvarů

```

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;
    }
}

```

- Zkušeným objektově orientovaným návrhářům jsou dobře známý možnosti, jak tohle obejít: například návrhový vzor VISITOR nebo dvojitě odbavení (dual dispatch). Ale použití téhoto technik není zadarmo a obvykle vedou ke struktuře procedurálního programu.

Opět zde vidíme komplementární povahu obou těchto definicí. Jsou to v podstatě protiklady! To ukazuje fundamentální dichotomii mezi objekty a datovými strukturami:

*Do procedurálního kódu (tj. kódu používajícího datové struktury) je jednoduché přidávat nové funkce bez změny existujících datových struktur. Na druhé straně do objektově orientovaného kódu je jednoduché přidávat nové třídy bez nutnosti změny existujících funkcí.*

Doplňkové tvrzení je rovněž pravdivé:

*Do procedurálního kódu se obtížně přidávají nové datové struktury, protože všechny funkce se musí změnit. Do objektově orientovaného kódu se obtížně přidávají nové funkce, protože se musí změnit všechny třídy.*

Takže to, co je obtížné pro objektové programování, je jednoduché pro procedurální programování, a co je obtížné pro procedurální programování, je jednoduché pro objektové programování!

V každém komplikovaném systému nastanou časy, kdy budeme chtít přidat nové datové typy, nikoli nové funkce. Objekty a objektové programování jsou pro tento účel nevhodnější. Na druhé straně nastanou i časy, kdy budeme chtít přidat nové funkce, jako protiklad k datovým typům. Pro takové případy je vhodnější procedurální kód a datové struktury. Zkušení programátoři vědí, že představa, že všechno je objekt, je mytus. Někdy opravdu chcete jednoduché datové struktury, které budou zpracovávat procedury.

## Démétřin zákon

Existuje jedna dobře známá heuristika, nazvaná *Démétřin zákon*<sup>2</sup>, která tvrdí, že modul by neměl nic vědět o vnitřku objektů, s nimiž pracuje. Jak jsme již viděli v minulé sekci, objekty svá data skrývají a zveřejňují operace. To znamená, že objekt by neměl zpřístupnit svou vnitřní strukturu pomocí přístupových objektů, protože pro vnitřní strukturu to znamená ji zveřejňovat a nikoliv skrývat.

Přesněji řečeno, Démétřin zákon říká, že metoda *f* třídy *C* by měla volat jen tyto metody:

- ◆ Metody třídy *C*
- ◆ Metody objektu vytvořeného metodou *f*
- ◆ Metody objektu předaného jako argument metodě *f*
- ◆ Metody objektu, který je instanční proměnnou třídy *C*

Metoda by neměla volat metody těch objektů, které vrací jakákoliv z povolených funkcí. Jiným slovy, rozmlouvej s přáteli, ne s cizinci.

Následující kód<sup>3</sup> zřejmě porušuje mimo jiné Démétřin zákon, protože volá funkci *getScratchDir()* po obdržení návratové hodnoty funkce *getOptions()* a pak volá funkci *getAbsolutePath()* po obdržení návratové hodnoty funkce *getScratchDir()*.

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

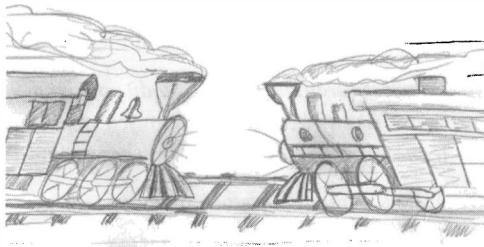
2. [http://en.wikipedia.org/wiki/Law\\_of\\_Demeter](http://en.wikipedia.org/wiki/Law_of_Demeter).

3. Nalezený někde v šabloně apache.

## Vykolejený vlak

Tento druh kódu se často nazývá *vykolejený vlak*, protože vypadá jako změř spojených vagonů. Sled podobných volání se obvykle považuje za nedbalý styl a měli bychom se mu vyhýbat [O36]. Nejlepším řešením je volání rozdělit, jak ukazuje následující kód:

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



Jsou tyto dvě části kódu porušením Démétrina zákona? Nadřízený modul ví, že objekt `ctxt` obsahuje volby, které obsahují pracovní adresář s absolutní cestou. To je pro jednu funkci poměrně dost informací. Volající funkce ví, jak procházet mnoho různých objektů.

Zda se jedná o porušení Démétrina zákona, závisí na tom, zda `ctxt`, `Options` a `ScratchDir` jsou objekty nebo datové struktury. Pokud to jsou objekty, jejich interní struktura by neměla být dostupná, ale skrytá, a pak znalost jejich vnitřků Démétrin zákon zcela jasně porušuje. Na druhé straně, pokud `ctxt`, `Options` a `ScratchDir` jsou datovými strukturami bez funkčnosti, pak je samozřejmě jejich interní struktura k dispozici a Démétrin zákon zde neplatí.

Použití přístupových funkcí tuto otázkou komplikuje. Pokud by byl kód napsán tak, jak je uvedeno zde, pak bychom se pravděpodobně na porušení Démétrina zákona neptali.

```
final String outputDir = ctxt.options.scratchDir.getAbsolutePath();
```

Tohle téma by bylo mnohem jednodušší, kdyby datové struktury měly prostě veřejné proměnné a neměly žádné funkce, zatímco objekty by měly privátní proměnné a veřejné funkce. Avšak existují rámcce a standardy (např. „beans“), které vyžadují, aby i jednoduché datové struktury měly přístupové i změnové metody.

## Hybridy

Tento zmatek vede často k nešťastným hybridním strukturám, které jsou napůl objekty a napůl datovými strukturami. Obsahují funkce, které dělají důležité věci, a také obsahují bud' veřejné proměnné, nebo veřejné přístupové a změnové metody, jež dělají z nejrůznějších důvodů ze soukromých proměnných veřejné a vedou ostatní externí funkce k tomu, aby tyto proměnné používaly takovým způsobem, jakým by používal procedurální program datovou strukturu.<sup>4</sup>

Takovéto hybridy ztěžují přidávání nových funkcí, ale také ztěžují přidávání nových datových struktur. Jsou tím nejhorším pro obě strany. Vyvarujte se jejich vytváření. Jsou příznakem zmatečného návrhu, jehož autoři si nejsou jisti – nebo ještě hůře, nevědí – zda potřebují mít zabezpečení před funkcemi nebo typy.

4. Tomu se někdy říká „chybějící schopnosti“ z [Refactoring].

## Skrytá struktura

Co když jsou `ctxt`, `options` a `scratchDir` objekty s reálnou funkčností? Protože se u objektů očekává skrytá vnitřní struktura, neměli bychom být schopni je procházet. Jak bychom se tedy dostali k absolutní cestě pracovního adresáře?

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

nebo

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

První možnost by mohla vést k explozi metod v objektu `ctxt`. Druhá předpokládá, že `getScratchDirectoryOption()` vráci datovou strukturu, ne objekt. Ani jedna možnost nevypadá dobře.

Pokud je `ctxt` objektem, měli bychom mu říci, aby *něco dělal*. Neměli bychom se ptát na jeho vnitřek. Takže proč jsme chtěli absolutní cestu k pracovnímu adresáři? Co s ní budeme dělat? Podivejte se na tento kód ze stejného modulu (o mnoho řádků níže):

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

Příměs různých úrovní detailu [O34][O6] činí trochu potíže. Tečky, lomítka, rozšíření souborů a objekty `File` by neměly být tak nedbale smíchané dohromady a promíchané s nadřazeným kódem. Když od toho ale odhlédneme, uvidíme, že záměrem získání absolutní cesty pracovního adresáře bylo vytvořit pracovní soubor daného názvu.

Takže, co kdybychom to řekli objektu `ctxt`, aby to provedl?

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

Zdá se, že pro objekt by to byl docela rozumný úkol! To umožňuje objektu `ctxt`, aby skryl svůj vnitřek a zabránil nynější funkci, aby porušila Démétřin zákon tím, že by procházela objekt, o kterém nemá nic vědět.

## Objekty pro přenos dat

Typickou formou datové struktury je třída bez funkcí a s veřejnými proměnnými. Tomu někdy říkáme objekt pro přenos dat neboli DTO (data transfer object). Tyto objekty jsou velmi užitečné struktury, zvláště při komunikaci s databázemi nebo rozbozech zpráv ze soketů atd. Často bývají na prvním místě v sekvenci fází překladu, které konvertují surová data v databázi do objektů v aplikačním kódu.

Poněkud obecnější je forma komponenty javabean, jak ukazuje výpis 6.7. Tyto komponenty mají soukromé proměnné, se kterými pracují přístupové metody. Zdá se, že jejich kvazizapouzdření některé puristické objektově orientované programátory potěší, ale obvykle nepřináší žádný další užitek.

**Výpis 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;  
    }  
}
```

## Aktivní záznam

Aktivní záznamy jsou speciálními formami objektů pro přenos dat (DTO). Jsou to datové struktury s veřejnými proměnnými, nebo s proměnnými, k nimž se přistupuje jako k datům v komponentách javabean, ale v typických případech mají navigační metody jako je `save` nebo `find`. Obecně tyto aktivní záznamy realizují přímý přenos dat z databázových tabulek nebo z jiných datových zdrojů.

Často bohužel zjišťujeme, že vývojáři zkouší zacházet s těmito datovými strukturami, jako by to byly objekty a vkládají do nich metody pro obchodní pravidla. To je nešikovné, protože to vytváří hybrid mezi datovou strukturou a objektem.

Řešením je samozřejmě brát aktivní záznam jako datovou strukturu a vytvořit samostatné objekty, které obsahují obchodní pravidla a jež skrývají svá interní data (což budou pravděpodobně instance aktivních záznamů).

## Shrnutí

Objekty zveřejňují funkčnosti a skrývají data. To zjednodušuje přidávání nových druhů objektů, aniž by se musela měnit stávající funkčnost. Rovněž to komplikuje přidávání nových funkčností do stávajících objektů. Datové struktury zveřejňují data a nemají žádnou signifikantní funkčnost. To zjednoduší přidávání nových funkčností ke stávajícím datovým strukturám, ale komplikuje přidávání nových datových struktur ke stávajícím funkcím.

V každém daném systému budeme někdy chtít přidávat pružně nové datové typy, a tudíž pro tuto část systému dáváme přednost objektům. Jindy budeme chtít pružně přidávat nové funkčnosti, a tudíž v této části systému budeme preferovat datové typy a procedury. Dobří softwaroví vývojáři rozumějí těmto otázkám bez předsudků a volí takový přístup, který je pro daný úkol nejlepší a je po ruce.

## Použitá literatura

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



# KAPITOLA 7

## Zpracování chyb

*Michael Feathers*

### V této kapitole najdete:

- ◆ Používejte výjimky raději než návratové kódy
- ◆ Pište nejdříve příkazy Try-Catch-Finally
- ◆ Používejte nekontrolované výjimky
- ◆ Poskytujte kontext s výjimkami
- ◆ Definujte třídy výjimek z hlediska potřeb volajícího
- ◆ Definujte normální tok
- ◆ Nevracejte hodnotu null
- ◆ Nepředávejte hodnotu null
- ◆ Závěr
- ◆ Literatura



Může se vám zdát divné mít sekci o zpracování chyb v knize, která pojednává o čistém kódu. Zpracování chyb je jen jednou z těch věcí, kterým se musíme během programování věnovat všichni. Vstupní hodnoty mohou být abnormální a zařízení se může porouchat. Stručně řečeno, věci se mohou ukázat, a když k tomu dojde, jako programátoři jsme zodpovědní za to, že nás kód bude dělat to, co má.

A však vztah k čistému kódu by měl být zřejmý. V mnoha kódech aplikace zpracování chyb naprostě převládá. Když říkám, že převládá, nemám na mysli, že zpracování chyb je to jediné, co dělají. Myslím tím to, že je téměř nemožné pochopit, co kód dělá, kvůli zpracování chyb, které je všudypřítomné. Zpracování chyb je důležité, ale když zatemňuje logiku, je špatné.

V této kapitole popíšu několik technik a úvah, které můžete použít pro psaní kódu, jenž je jak čistý, tak i robustní – kód, který zpracuje chyby s elegancí a šarmem.

## Používejte výjimky raději než návratové kódy

V dávné minulosti bylo mnoho jazyků, které neměly výjimky. V těchto jazycích byly techniky zpracování a informování o chybách omezené. Kód mohl buď nastavit příznak chyby, nebo vrátit číslo chyby, které mohl volající objekt prověřit. Tento pohled kód ilustruje ve výpisu 7.1.

### Výpis 7.1. DeviceController.java

```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Zkontrolujte stav zařízení
        if (handle != DeviceHandle.INVALID) {
            // Uložte stav zařízení do pole záznamu
            retrieveDeviceRecord(handle);
            // pokud není pozastaveno, vypnout
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " + DEV1.toString());
        }
    }
    ...
}
```

Problém tohoto pohledu spočívá v tom, že zaneráduje volající objekt. Ten musí prověřit chyby bezprostředně po zavolání. Bohužel se na to dá jednoduše zapomenout. Z tohoto důvodu je lepší, když se při výskytu chyby vyvolá výjimka. Volající kód je čistější. Jeho logika není zaměložována zpracováním chyb.

Výpis 7.2 ukazuje kód poté, co jsme si zvolili možnost vyvolávat výjimky v metodách, které mohou detekovat chyby.

**Výpis 7.2.** DeviceController.java (s výjimkami)

```
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("Invalid handle for: " + id.toString());  
        ...  
    }  
    ...  
}
```

Všimněte si, oč je kód čistší. Není to pouze záležitost estetiky. Kód je lepší, protože dvě předtím propletené záležitosti, algoritmus pro odstavení zařízení a pro zpracování chyb, jsou od sebe nyní odděleny. Můžete se na obě věci podívat a chápout je nezávisle.

## Pište nejdříve příkazy Try-Catch-Finally

Jednou z nejzajímavějších věcí u výjimek je, že uvnitř vašeho programu *definují rozsah platnosti identifikátorů*. Když provádíté kód v části `try` příkazu `try-catch-finally`, určujete tím, že provádění programu může být přerušeno ve kterémkoliv bodě a může pokračovat ve větví `catch`.

Z tohoto pohledu se bloky `try` podobají transakcím. Část `catch` musí zanechat váš program v konsistentním stavu bez ohledu na to, co se stane v části `try`. Z tohoto důvodu je dobré začít s příkazem `try-catch-finally`, pokud píšete kód, který může generovat výjimky. To vám pomůže definovat, co by měl uživatel tohoto kódu očekávat, bez ohledu na to, jaká chyba nastane v kódu, který se provádí v části `try`.

Podívejme se na jeden příklad. Potřebujeme napsat kód, který přistupuje k souboru a čte nějaké serializované objekty.

Začneme u jednotkového testu, který ukazuje, že obdržíme výjimku, pokud soubor neexistuje:

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

Test nás přivede k tomu, abychom vytvořili tento prázdný kód:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
    // prázdná, vrácená hodnota, dokud nebudeme mít skutečnou implementaci
    return new ArrayList<RecordedGrip>();
}
```

Náš test neprojde, protože nevyvolá výjimku. V dalším kroku změníme jeho implementaci tak, aby se pokusila přistoupit k neplatnému souboru. Tato operace vygeneruje výjimku:

```
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 nyní prochází, protože jsme výjimku zachytili. V tomto okamžiku můžeme refaktorovat. Můžeme zúžit typ výjimky, kterou zachytáváme, aby odpovídala typu, jenž se ve skutečnosti generuje v konstruktoru `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>();
}
```

Když jsme nyní definovali obor ve struktuře příkazu `try-catch`, můžeme použít TDD (test-driven development), abychom napsali zbytek logiky, kterou potřebujeme. Tato logika bude zařazena mezi vytvoření `FileInputStream` a `close` a může předstírat, že je vše v pořádku.

Zkuste napsat testy, které si vynucují výjimky, a poté přidejte do obslužné rutiny funkčnost tak, aby testy prošly. To způsobí, že nejdříve vytvoříte transakční obor bloku `try` a pomůže vám to udržovat transakční povahu této části.

## Používejte nekontrolované výjimky

Tato debata je již za námi. Již léta debatovali programátoři v Javě o výhodách a nevýhodách kontrolovaných výjimek. Když byly kontrolované výjimky poprvé představeny v první verzi Javy, zdálo se, že jde o výbornou myšlenku. Všechny typy výjimek by byly uvedeny v signatuře každé metody, které může předávat volajícímu. Dále byly tyto výjimky součástí typu metody. Váš kód by se vůbec nepřeložil, pokud by signatura nevyhovovala tomu, co může kód dělat.

Tehdy jsme mysleli, že kontrolované výjimky jsou dobrým nápadem, a skutečně mohou *nějaké* výhody mít. Avšak nyní je zřejmé, že pro tvorbu robustního softwaru nejsou nezbytné. Jazyk C# kontro-

lované výjimky nemá a navzdory chrabrému pokusům je neobsahuje ani jazyk C++. Python a Ruby také ne. Přesto je možné psát ve všech těchto jazyčích robustní software. Protože právě o tohle jde, musíme se rozhodnout – skutečně – zda se používání kontrolovaných výjimek vyplatí.

Jaká je jejich cena? Cenou za používání kontrolovaných výjimek je porušení principu otevřenosti a uzavřenosti<sup>1</sup>. Pokud v kódu některé metody generujete kontrolovanou výjimku, a příkaz `catch` je tří o úrovni výše, *musíte tuto výjimku deklarovat v signatuře všech metod mezi místem jejího vzniku a klauzulí catch*. To znamená, že změna na nízké úrovni softwaru může vyvolat změny v signaturách na více vyšších úrovních. Změněné moduly je nutné znovu sestavit a nainstalovat, i když se jich změny vůbec netýkají.

Podívejte se na hierarchii volání ve velkém systému. Funkce, které jsou nahoře, volají funkce, jež jsou níže. Ty volají další nižší funkce a tak to jde do nekonečna. Řekněme, že jedna z nejnižších funkcí se změní takovým způsobem, že musí generovat výjimku. Jestliže jde o kontrolovanou výjimku, pak je nutné přidat do signatury funkce klauzuli `throws`. Ale to znamená, že každá funkce, která volá naši změněnou funkci, se také musí změnit. Bud' musí zachytávat novou výjimku, nebo musíme do její signatury přidat příslušnou klauzuli `throws`. Do nekonečna. Čistým výsledkem je kaskáda změn, které si razí cestu od nejnižších úrovni softwaru do těch nejvyšších! Zapouzdření se poruší, protože všechny funkce v cestě od klauzule `throw` musí znát detaily o výjimce nízké úrovni. Za předpokladu, že smyslem výjimek je umožnit zpracování chyb „na dálku“, je ostudou, že kontrolované výjimky tímto způsobem dokážou zapouzdření porušit.

Kontrolované výjimky mohou být někdy užitečné, pokud píšete nějakou důležitou knihovnu: Musíte je zachytit. Ale během vývoje obecné aplikace je cena za závislost vyšší, než je jejich přínos.

## Poskytujte kontext s výjimkami

Každá výjimka, kterou vygenerujete, by měla poskytnout dostatek souvisejících informací, aby bylo možné určit zdroj a místo chyby. V Javě můžete mít k dispozici ke každé chybě trasování zásobníku. Ale trasování zásobníku vám nemůže poskytnout záměr operace, která se nezdařila.

Vytvářejte informativní chybová hlášení a předávejte je spolu s vašimi výjimkami. Uveďte operaci, která se nezdařila, a druh selhání. Pokud běh vaší aplikace protokolujete, předávejte dostatečné informace, abyste byli schopni chybu zaprotokolovat v klauzuli `catch`.

## Definujte třídy výjimek z hlediska potřeb volajícího

Existuje mnoho způsobů klasifikace chyb. Můžeme je klasifikovat podle jejich zdroje: přicházejí z té, nebo oné komponenty? Nebo podle jejich typu: jde o selhání zařízení, sítě, nebo jde o programovou chybu? Jestliže však v aplikaci definujeme třídy výjimek, měla by naším nejdůležitějším úkolem být znalost, jak byly zachyceny.

Podívejme se na příklad špatné klasifikace chyb. Máme zde příkaz `try-catch-finally`, která volá knihovnu třetí strany. Zahrnuje všechny výjimky, které může volání způsobit:

1. [Martin].

```

ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}

```

Tento příkaz obsahuje mnoho zdvojení a neměli bychom tím být překvapeni. Ve většině případů zpracování výjimek je prováděná činnost relativně standardní bez ohledu na skutečný důvod. Musíme zaznamenat chybu a ověřit si, že můžeme pokračovat.

V tomto případě, protože víme, že prováděná činnost je pro všechny výjimky zhruba stejná, můžeme svůj kód značně sjednodušit, když ho zabalíme do API, které voláme, a zajistíme, aby návratovou hodnotou byl obecný typ výjimky:

```

LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}

```

Třída `LocalPort` je pouze jednoduchou obálkou, která zachytává a překládá výjimky generované třídou `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);
        }
    }
    ...
}

```

Obálka, kterou jsme definovali pro třídu `ACMEPort`, může být velmi užitečná. Obalování API třetí strany je skutečně tím nejlepším postupem. Když vytváříte obal pro API třetí strany, minimalizujete tím svou závislost na něm: v budoucnu si můžete bez nějakých větších důsledků zvolit jinou knihovnu. Obalování rovněž zjednoduší napodobování volání objektů třetí strany při testování vašeho vlastního kódu.

Jednou z posledních výhod obalování je, že nejste při návrhu vázáni na volbu rozhraní API nějakého konkrétního dodavatele. Můžete definovat API, se kterým se vám bude pohodlně pracovat. V předchozím příkladu jsme definovali jediný typ výjimky pro selhání zařízení port a zjistili jsme, že bychom mohli psát mnohem čistší kód.

Samostatná třída pro výjimky je často dobrá pro určitou oblast kódu. Informace, která se posílá spolu s výjimkou, může chyby dále charakterizovat. Používejte jiné třídy jen v těch případech, kdy potřebujete zachytit jednu výjimku a ostatní nechat být.

## Definujte normální tok

Pokud se budete řídit radami z předchozích sekcí, bude výsledkem dobré oddělení kódu obchodní logiky od kódu pro zpracování chyb. Značná část kódu bude vypadat jako čistý a nevyumělkovaný algoritmus. Avšak proces jeho vytváření vytlačuje detekci chyb na okraj vašeho programu. Zabalíte externí API tak, aby mohlo generovat vaše vlastní výjimky, a nad vaším kódem definujete obslužnou rutinu tak, abyste mohli zpracovat jakýkoliv předčasně ukončený chod programu. Většinou je tento přístup velmi dobrý, ale existují případy, kdy běh předčasně ukončit nechcete.



Podívejme se na jeden příklad. Máme zde jakýsi nešikovný kód, který sečítá náklady v nějaké účetní aplikaci:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

V tomto obchodě platí, že pokud jsou jídla účtována do nákladů, stávají se součástí celkové sumy. Pokud ne, dostává v ten den zaměstnanec částku za jídlo *per diem*. Výjimka vnáší do logiky zmatek. Nebylo by lepší, kdybychom nemuseli zpracovávat jeden speciální případ? Pokud bychom nemuseli, vypadal by nás kód mnohem jednodušeji. Vypadal by takto:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Můžeme kód takto zjednodušit? Zdá se, že ano. Můžeme změnit metodu `ExpenseReportDAO` tak, aby vždy vracela objekt `MealExpense`. Pokud žádné náklady na jídlo neexistují, vrací objekt `MealExpense`, který vrací hodnotu *per diem* jako celkovou sumu:

```
public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
        // vrácení implicitní hodnoty per diem
    }
}
```

Tohle se nazývá objekt pro zvláštní případ (Special Case Pattern – [Fowler]). Vytvoříte třídu nebo konfigurujete objekt tak, aby pro vás ošetřil zvláštní případy. Když to uděláte, klientský kód se nemusí zabývat funkčností pro výjimky. Tato funkčnost je zapouzdřena v objektu určeném pro zvláštní případy.

## Nevracejte hodnotu null

Jsem toho názoru, že jakákoliv diskuse o ošetřování chyb by měla obsahovat zmínku o tom, jak chyby vyvoláváme. Na prvním místě je na seznamu vracení hodnoty `null`. Viděl jsem bezpočet aplikací, ve kterých téměř každý druhý řádek kontroloval hodnotu `null`. Zde máme příklad takového kódu:

```
public void registerItem(Item item) {
    if (item != null) {
        ItemRegistry registry = persistentStore.getItemRegistry();
        if (registry != null) {
            Item existing = registry.getItem(item.getID());
            if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
            }
        }
    }
}
```

Pokud pracujete s podobným kódem aplikace, nemusí se vám to zdát tak špatné, ale špatné to je! Pokud vracíme hodnotu `null`, v podstatě si přidáváme další práci a vytváříme problémy volajícím objektům. Přitom stačí jedna chybějící kontrola hodnoty `null` a celá aplikace se vymkne kontrole.

Všimli jste si, že ve druhém řádku vnořeného příkazu `if` chybí kontrola na `null`? V době běhu programu bychom obdrželi výjimku typu `NullPointerException`, ať už tuto výjimku na vysoké úrovni někdo zachytí nebo ne, obojoj je špatně. Jak byste měli přesně reagovat na výjimku typu `NullPointerException`, která vznikla někde v hlubinách vaší aplikace?

Je jednoduché říci, že problém nadřazeného kódu je v tom, že chybí kontrola hodnoty `null`, ale ve skutečnosti je problém v tom, že jich je příliš mnoho. Pokud jste v pokušení vracet z metody `null`, zvažte místo toho vyvolání výjimky nebo vrácení objektu pro zvláštní případ (SPECIAL CASE OBJECT). Pokud voláte metodu API třetí strany, která vrací hodnotu `null`, zvažte obalení této metody tak, aby buď vyvolala výjimku, nebo vrátila objekt pro zvláštní případ.

V mnoha případech představují objekty pro zvláštní případ jednoduché řešení. Představte si, že máte kód, jako je tento:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

Právě zde může metoda vrátit hodnotu `null`, ale musí? Pokud změníme metodu `getEmployee` tak, aby vracela prázdný seznam, můžeme kód vyčistit takto:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

Naštěstí obsahuje Java metodu `Collections.emptyList()`, jež vrací předdefinovaný neměnný seznam, který můžeme použít pro nás účel:

```
public List<Employee> getEmployees() {
    if( .. there are no employees .. )
        return Collections.emptyList();
}
```

Budete-li psát kód tímto způsobem, minimalizujete možnost výjimek typu `NullPointerException` a vás kód bude čistší.

## Nepředávejte hodnotu null

Vracení hodnoty `null` je špatné, ale předávání této hodnoty je ještě horší. Pracujete-li s API, které předpokládá, že budete předávat `null`, měli byste se tomu ve vašem kódu vyhnout všude, kde je to možné.

Podívejme se na tento příklad, který ukazuje proč. Máme zde jednoduchou metodu, která počítá metriku dvou bodů:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

Co se stane, když někdo předá jako argument hodnotu `null`?

```
calculator.xProjection(null, new Point(12, 13));
```

Obdržíme samozřejmě výjimku typu `NullPointerException`.

Jak to můžeme opravit? Mohli bychom vytvořit nový typ výjimky a vygenerovat ji:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
            throw new InvalidArgumentException(
                "Invalid argument for MetricsCalculator.xProjection");
        }
        return (p2.x - p1.x) * 1.5;
    }
}
```

Je to lepší? Mohlo by to být o něco lepší, než je výjimka způsobená hodnotou `null`, ale mějte na paměti, že potřebujeme definovat obslužnou rutinu pro výjimku typu `InvalidArgumentException`. Co by měla tato rutina dělat? Je na tomto postupu vůbec něco dobrého?

Existuje i jiná alternativa. Mohli bychom použít několik asercí:

```
public class MetricsCalculator {
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 should not be null";
        assert p2 != null : "p2 should not be null";
        return (p2.x - p1.x) * 1.5;
    }
}
```

Je to dobré k dokumentaci, ale problém to neřeší. Pokud někdo předá hodnotu `null`, stále to bude znamenat chybu za běhu programu.

Ve většině programovacích jazyků neexistuje dobrý způsob, jak se vypořádat s hodnotou `null`, kterou volající objekt předává náhodně. Protože se jedná o nás případ, je racionálním přístupem jako standard zakázat předávání `null`. Když to provedete, můžete psát kód s tím, že hodnota `null` v seznamu argumentů indikuje problém a výsledkem bude mnohem menší množství chyb vzniklých nedbalostí.

## Závěr

Čistý kód je čitelný, ale musí být zároveň robustní. Tohle nejsou protikladné cíle. Můžeme psát robustní a čistý kód, pokud spatřujeme ve zpracování chyb samostatnou záležitost, něco, co lze vnímat nezávisle na hlavní logice kódu. Když se nám to podaří, můžeme o tom argumentovat nezávisle a udělat v udržovatelnosti našeho kódu velký pokrok.

## Použitá literatura

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

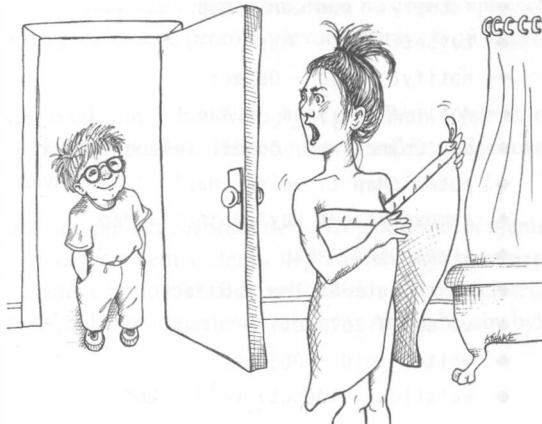
# KAPITOLA 8

## Hranice

*James Grenning*

### V této kapitole najdete:

- ◆ Použití kódu třetí strany
- ◆ Zkoumání a studium hranic
- ◆ Studium log4j
- ◆ Poznávací testy se vyplatí
- ◆ Používání kódu, který zatím neexistuje
- ◆ Čisté hranice
- ◆ Literatura



Jen zřídkakdy máme v našich systémech veškerý software pod kontrolou. Někdy kupujeme balíčky třetí strany nebo používáme otevřený zdrojový kód. Jindy jsme závislí na ostatních týmech v naší společnosti, které pro nás vytvářejí komponenty nebo subsystémy. Tento cizí software musíme bezchybně integrovat do našeho. V této kapitole se podíváme na postupy a techniky, které mají zajistit, aby se na hranicích našeho softwaru nevyskytovaly chyby.

## Použití kódu třetí strany

Mezi poskytovatelem rozhraní a jeho uživatelem existuje jakési přirozené napětí. Poskytovatelé balíčků třetích stran a aplikačních rámců usilují o širokou využitelnost, aby mohli oslovit širokou veřejnost, která používá mnoho různých prostředí. Na druhé straně uživatelé chtějí rozhraní, které vyhovuje jejich praktickým potřebám. Toto napětí může způsobovat problémy na hranicích našich systémů.

Podívejme se například na rozhraní `java.util.Map`. Jak můžete vidět na obrázku 8.1, jsou tato rozhraní značně rozsáhlá a disponují velkým množstvím schopností. Tyto schopnosti a pružnost jsou určitě užitečné, ale může to být také nevýhoda. Aplikace může například vytvořit rozhraní `Map` a předávat je dále. Našim úmyslem by mohlo být, aby v něm žádný z jeho příjemců nemohl nic smazat. Ale hned na začátku seznamu je metoda `clear()`. Kterýkoliv jeho uživatel má možnost mazat. Nebo mohou být konvence návrhutakové, že v objektu implementujícím rozhraní `Map` lze ukládat jen určité typy objektů, ale omezení tohoto rozhraní při ukládání různých typů objektů není spolehlivé. Když bude uživatel chtít, může do něj vkládat položky jakéhokoliv druhu.

- ◆ `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`

Obrázek 8.1. Metody rozhraní `Map`

Pokud naše aplikace potřebuje Map spolu se Sensory, mohli byste je nastavit takto:

```
Map sensors = new HashMap();
```

Později, když bude nějaká jiná část kódu požadovat přístup k senzorům, uvidíte následující kód:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

S tím se nesetkáme jen jednou, ale mnohokrát a ve všech částech kódu. Klient tohoto kódu nese zodpovědnost za to, že z rozhraní Map dostane Object a převede jej na správný typ. To funguje, ale není to čistý kód. Kód k nám také není natolik sdílný, jak by mohl. Čitelnost kódu lze o hodně zlepšit použitím generických typů, jak uvidíte dále:

```
Map<Sensor> sensors = new HashMap<Sensor>();  
...  
Sensor s = sensors.get(sensorId);
```

Tím se však neřeší problém, že Map<Sensor> toho poskytuje více, než potřebujeme nebo chceme.

Předávání instance typu Map<Sensor> libovolně po celém systému znamená, že pokud se rozhraní Map někdy změní, kód bude třeba opravovat na mnoha místech. Možná si řeknete, že taková změna není pravděpodobná, ale vzpomeňte si, že k podobné změně již došlo, když byla do Javy 5 přidaná podpora genericity. Opravdu jsme viděli systémy, kde bylo použití generických typů zakázáno právě z důvodu velkého množství změn, které jsou cenou za volné používání rozhraní Map.

Lepší způsob jeho použití by mohl vypadat následovně. Žádný uživatel Sensorů by se nestaral o to, zda se generické typy používají. Tato volba se stala (a vždy by měla být) implementačním detailem.

```
public class Sensors {  
    private Map sensors = new HashMap();  
    public Sensor getById(String id) {  
        return (Sensor) sensors.get(id);  
    }  
    //část kódu  
}
```

Rozhraní na hranici (Map) je skryto. Může se rozvinout bez velkých důsledků na zbytek aplikace. Použití generických typů již není žádným velkým problémem, protože převod a správa typů se provádí uvnitř třídy Sensors.

Toto rozhraní je i šito na míru aplikace a omezeno tak, aby vyhovovalo jejím potřebám. Výsledkem je srozumitelnější kód s menším rizikem nesprávného použití. Třída Sensors může podporovat vytváření obchodních pravidel a pravidel pro návrh.

Nedoporučujeme, abyste takto zapouzdřovali každé použití rozhraní Map. Naopak doporučujeme, abyste je (nebo kterékoliv jiné rozhraní na hranici) ve svém systému nepředávali jako parametr. Pokud používáte hraniční rozhraní, jako je Map, mějte ho uvnitř třídy nebo blízko příbuzných tříd, kde se používají. Vyhněte se tomu, abyste je předávali jako argumenty veřejným API nebo aby byla z API vracena.

## Zkoumání a studium hranic

Kód třetí strany nám umožnuje získat více funkcionality za kratší čas. Chceme-li začít používat nějaký balíček třetí strany, kde začít? Není naším úkolem jejich kód otestovat, ale v našem nejvlastnějším zájmu je napsat testy pro kód třetí strany, který budeme používat.

Předpokládejme, že nevíme, jak máme knihovnu třetí strany používat. Mohli bychom věnovat den nebo dva (či více) čtení dokumentace a pak o způsobu jejího použití rozhodnout. Pak bychom mohli napsat vlastní kód s kódem třetí strany a ověřit si, zda dělá to, co předpokládáme. Nebylo by nic překvapujícího, kdybychom se zapletli do osidel zdlouhavé ladící procedury, která má jeden účel: Zjistit, zda chyby, na které narázíme, jsou naše nebo jejich.

Studium kódu třetí strany je těžké. Integrování kódu třetí strany je také těžké. Dělat obojí najednou je dvojnásobně těžké. Co kdybychom to zkusili jinak? Místo experimentování a zkoušení nových věcí bychom mohli napsat v ostrém kódu nějaké testy a s jejich pomocí si ověřit, nakolik jsme schopni mu rozumět. Jim Newkirk nazývá takové testy *poznávací testy*<sup>1</sup>. V těchto testech voláme API třetí strany tak, jak očekáváme, že je bude používat naše aplikace. V podstatě jde o kontrolní experimenty, které ověřují, zda jsme pochopili, jak správně používat rozhraní API. Testy se zaměřují na to, co chceme z API dostat.

## Studium log4j

Dejme tomu, že chceme použít balíček apache log4j k protokolování a dáme mu přednost před vlastním uživatelským protokolovacím nástrojem. Stáhneme si jej a otevřeme si úvodní stránku jeho dokumentace. Bez nějakého velkého čtení si napíšeme první testovací příklad a očekáváme, že vypíše na konzolu pozdrav „hello“.

```
@Test
public void testLogCreate() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.info("hello");
}
```

Když kód spustíme, protokolovací nástroj vyvolá chybu, která nám oznámí, že potřebujeme něco, co se nazývá Appender. Po dalším čtení zjištujeme, že existuje ConsoleAppender. Vytvoříme tedy jeho instanci a podíváme se, zda jsme tajemství protokolování s výstupem na konzolu vyřešili.

```
@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info("hello");
}
```

Tentokrát zjistíme, že Appender nemá žádný výstupní datový proud. Špatné – zdá se logické, že by jej měl mít. Když najdeme nějaké informace na Googlu, zkusíme následující:

1. [BeckTDD], pp. 136–137.

```
@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("hello");
}
```

Tento kód funguje a protokolová zpráva obsahující „hello“ se na konzole skutečně objeví! Zdá se divné, že musíme říci objektu `ConsoleAppender`, aby psal na konzolu.

Je zajímavé, že i když odstraníme argument `ConsoleAppender.SystemOut`, řetězec „hello“ se vytiskne. Ale když odebereme `PatternLayout`, opět uvidíme chybová hlášení, že postrádáme výstupní datový proud. Je to velmi podivné chování.

Když se podíváme do dokumentace trochu pečlivěji, zjistíme, že implicitní konstruktor objektu `ConsoleAppender` není „konfigurován“, což nebývá samozřejmé a ani to nebude užitečné. Zdá se, že balíček `log4j` obsahuje chybu nebo minimálně nekonzistenci.

Po dalším hledání na Googlu, čtení a testování dospejeme nakonec k výpisu 8.1. O tom, jak funguje `log4j`, jsme zjistili dost informací a tuto znalost jsme zakódovali to několika jednoduchých jednotkových testů.

#### Výpis 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");
    }
}

```

Nyní už víme, jak inicializovat jednoduchý konzolový protokolovací nástroj, a tuto znalost můžeme zapouzdřít do vlastní protokolovací třídy tak, aby byl zbytek aplikace od hraničního rozhraní `log4j` izolován.

## Poznávací testy se vyplatí

Poznávací testy nás nakonec nic nestály. API jsme se museli naučit tak jako tak a psaní testů bylo jen jednoduchým a samostatným způsobem, jak se k této znalosti dostat. Poznávací testy byly precizními experimenty, které nám pomohly kód pochopit.

Nejen že jsou poznávací testy zadarmo, ale dokonce přináší pozitivní zisk z investice. Když vyjdou nové verze balíčků třetí strany, spustíme poznávací testy, abychom zjistili, zda jsou v jejich funkčnosti nějaké rozdíly.

Poznávací testy ověřují, zda používané balíčky třetí strany fungují tak, jak očekáváme. Jakmile je jedenou kód třetí strany do našeho kódu integrován, nejsou žádné záruky, že bude našim potřebám i nadále vyhovovat. Původní autoři budou nuteni kód měnit podle svých potřeb. Budou opravovat chyby a přidávat nové schopnosti. S každou novou verzí budou přicházet nová rizika. Pokud se balíček třetí strany změní jakýmkoliv způsobem, který testům nevyhoví, okamžitě na to přijdeme.

Ať už se potřebujete učit pomocí poznávacích testů či nikoliv, měla by existovat jasná hranice, podporovaná sadou výstupních testů, které používají rozhraní tímto způsobem, jakým to dělá ostrý kód. Bez těchto *hraničních testů*, které mají usnadnit migraci kódu, bychom mohli být v pokušení používat starou verzi déle, než by bylo vhodné.

## Používání kódu, který zatím neexistuje

Existuje jiný druh hranice, který odděluje známé od neznámého. Často se v kódu objevují místa, kde máme pocit, že jsme se svými znalostmi vedle. Někdy se jedná o věci, které jsou na druhé straně hranice a jsou nepoznatelné (alespoň v daném okamžiku). Někdy se sami rozhodneme, že tím, co je za touto hranicí, se nebudeme zbývat.

Před několika lety jsem byl členem týmu, který využíval software pro rádiový komunikační systém. Byl tam subsystém „Transmitter“, o kterém jsme nevěděli téměř nic a lidé, již za něj zodpovídali, se nedostali k tomu, aby rozhraní definovali. Nechtěli jsme žádné zdržení a tak jsme začali pracovat na kódu, který byl na mile vzdálený tomu, co jsme neznali.

Měli jsme celkem dobrou představu, kde končí náš svět a kde začíná nový. V průběhu práce jsme čas od času na tuto hranici narazili. Ačkoliv náš pohled za ní zatemňovala mlha neznalosti, naše práce nás donutila si uvědomit, co jsme po hraničním rozhraní *chtěli*. Chtěli jsme vysílači říci přibližně tohle:

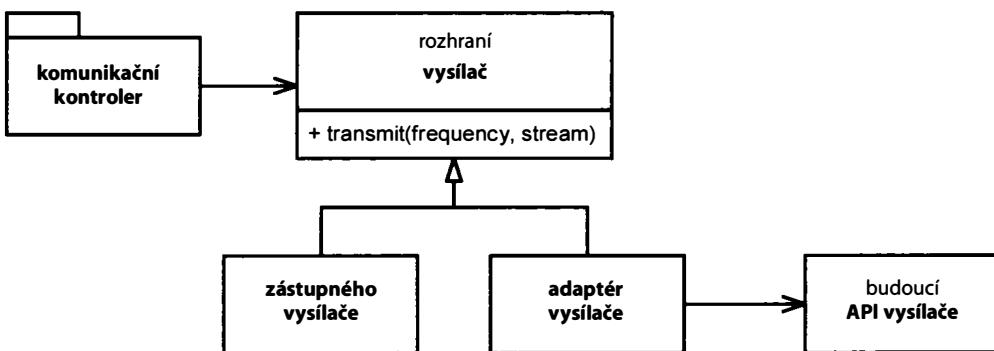
*Naladte vysílač na danou frekvenci a v analogové formě vysílejte data pocházející z tohoto proudu.*

Neměli jsme ponětí o tom, jak se to bude dělat, protože příslušné části API dosud nebyly navrženy. Tak jsme se rozhodli se věnovat detailům později.

Aby nás nic nezdržovalo, definovali jsme si vlastní rozhraní. Zvolili jsme si snadno zapamatovatelný název, něco jako `Transmitter`. Zavedli jsme metodu `transmit`, která dostala frekvenci a datový proud. To bylo rozhraní, které jsme si *přáli* mít.

Na psaní rozhraní, které jsme chtěli, byla jedna věc pozitivní. Měli jsme jej pod kontrolou. To umožňuje, aby byl klientský kód čitelnější a lépe zaměřený na to, co má dělat.

Na obrázku 8.2 můžete vidět, že jsme izolovali třídy `CommunicationsController` od API vysílače (který nemáme pod kontrolou a jenž není definován). Při použití vlastního specifického aplikačního rozhraní bude kód třídy `CommunicationsController` čistý a expresivní. Jakmile bylo API vysílače definováno, napsali jsme `TransmitterAdapter` a vyplnili existující mezeru. Vzor ADAPTÉR<sup>2</sup> zapouzdří interakci s API a umožňuje, aby se v případě změn v API prováděly opravy kódu jen na jednom místě.



Obrázek 8.2. Pravděpodobné určení funkce vysílače

Tento návrh nám rovněž poskytuje v kódu pro testování velmi užitečný šep<sup>3</sup>. S použitím vhodného zástupného vysílače, třídy `FakeTransmitter`, můžeme testovat třídy z balíčku `CommunicationsController`. Můžeme rovněž vytvořit hraniční testy, jakmile máme rozhraní `TransmitterAPI`, které nám prověří, zda používáme API správně.

## Čisté hranice

Na hranicích se dějí zajímavé věci. Jednou z nich jsou změny. Dobrý software se dokáže přizpůsobit změnám bez velkých investic a předělávání. Jestliže používáme kód, který nemáme pod kontrolou, musíme se velmi pečlivě postarat, abychom své investice chránili a pojistili si, že změny nebudou v budoucnu příliš drahé.

Kód na hranicích vyžaduje jasnou separaci a testy, které definují to, co očekáváme. Měli bychom se vyhnout tomu, aby náš kód věděl o zvláštnostech kódu třetí strany příliš mnoho. Lépe je záviset

2. Viz vzor Adaptér v [GOF].

3. Další informace o švech naleznete v [WELC].

na něčem, co máte pod kontrolou než na něčem, co pod kontrolou nemáte, aby to nakonec nemělo pod kontrolou vás.

Hranice třetí strany zvládáme tak, že máme v kódu jen několik míst, která na ně odkazují. Můžeme je zabalit, jak jsme to udělali s rozhraním Map, nebo můžeme použít vzor ADAPTÉR pro konverzi mezi naším kompletním rozhraním a rozhraním poskytovaným třetí stranou. V každém případě je vůči nám kód sdílnější, podporuje vnitřně po celé hranici interně konzistentní používání rozhraní a z hlediska údržby obsahuje méně míst, ve kterých se může kód třetí strany měnit.

## Použitá literatura

[BeckTDD]: *Test Driven Development*, Kent Beck, Addison-Wesley, 2003.

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

[WELC]: *Working Effectively with Legacy Code*, Addison-Wesley, 2004.

# KAPITOLA 9

## Jednotkové testy

### V této kapitole najdete:

- ◆ Tři zákony vývoje řízeného testy (TDD)
- ◆ Mějte testy čisté
- ◆ Čisté testy
- ◆ Jedna aserce na jeden test
- ◆ F.I.R.S.T.
- ◆ Závěr
- ◆ Použitá literatura



Náš obor udělal během posledních deseti let značný pokrok. V roce 1997 nikdo neslyšel o vývoji řízeném testy (TTD – Test Driven Development). Pro drtivou většinu z nás byly jednotkové testy jakési krátké kousky kódu na jedno použití, které jsme napsali jen proto, abychom si ověřili, že naše programy „fungují“. Psali jsme pilně třídy a metody a pak jsme vymysleli nějaký jednoúčelový kód, abychom si je otestovali. V typickém případě to znamenalo nějaký druh jednoduchého ovladače, který nám umožnil vstupovat ručně do programu, jejž jsme napsali.

Vzpomínám si, když jsem v polovině devadesátých let psal program v C++, který v sobě zahrnoval systém pracující v reálném čase. Program by jednoduchý časovač s následující signaturou:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

Myšlenka byla jednoduchá. Metoda `execute` třídy `Command` poběží v novém podprocesu, jakmile uplyne předem stanovený počet milisekund. Problém byl, jak jej testovat.

Napsal jsem ve spěchu jednoduchý ovladač, který snímal znaky z klávesnice. Pokaždé, když byl zadán nějaký znak, naplánoval příkaz, který za pět sekund zadal týž znak. Poté jsem na klávesnici vytuhal rytmickou melodii a čekal jsem, až ji po pěti sekundách uvidím na obrazovce.

“I . . . want-a-girl . . . just . . . like-the-girl-who-marr . . . ied . . . dear . . . old . . . dad.”

Tuto melodii jsem si skutečně zpíval, když jsem vkládal znak „ a pak jsem si ji zpíval znovu, když se tečky objevily na obrazovce.

To byl můj test! Když jsem ukázal program kolegům a viděl, že funguje, testy jsem zahodil.

Jak jsem již řekl, náš obor udělal značný pokrok. Dnes bych napsal testy, které by zaručovaly, že každý roh a každá skulina takového kódu funguje tak, jak očekávám. Oddělil bych kód od operačního systému, místo abych volal jen několik standardních časovacích funkcí. Ty bych simuloval, abych nad nimi měl během práce absolutní kontrolu. Naplánoval bych příkazy, které nastavují logické příznaky, a pak bych posunul čas dopředu, abych měl jistotu, že tyto příznaky přešly z hodnoty false na hodnoty true, hned jak jsem nastavil čas na správnou hodnotu.

Jakmile bych měl připravenou sadu testů, přesvědčil bych se, že tyto testy jsou vhodné pro kohokoliv, kdo by potřeboval s tímto kódem pracovat. Postaral bych se, aby byly tyto testy zahrnuty do stejného zdrojového balíčku.

Ano, udělali jsme značný pokrok, ale musíme pokračovat dále. Hnutí agilního programování a TDD povzbudily řadu programátorů k psaní automatizovaných jednotkových testů a řady těchto programátorů se rozšiřují každý den. Ale v rámci divokého shonu, jak zavést do našeho oboru testování, nepochopilo mnoho programátorů některé rafinované, ale důležité zásady, jak psát dobré testy.

## Tři zákony vývoje řízeného testy (TDD)

Nyní již každý ví, že metoda vývoje řízeného testy nás vede k napsání jednotkových testů ještě před psaním ostrého kódu. Ale tohle pravidlo je jen špičkou ledovce. Podívejte se na následující tři zákony<sup>1</sup>:

1. *Professionalism and Test-Driven Development*, Robert C. Martin, Object Mentor, IEEE Software, květen/červen 2007 (svazek. 24, č. 3) str. 32–36.  
<http://doi.ieeecomputersociety.org/10.1109/MS.2007.85>.

**První zákon:** Nebudete psát ostrý kód, dokud nebudete mít napsané jednotkové testy, které budou selhat.

**Druhý zákon:** Nepište více jednotkových testů, než ty, které momentálně neprojdou. Nelze-li je přeložit, neprošly.

**Třetí zákon:** Nesmíte psát další ostrý kód, než ten, který je třeba, aby stávající nevyhovující testy prošly.

Tyto tři zákony vás uzavřou do cyklu, který je možná dlouhý jen třicet sekund. Testy a ostrý kód se tvoří *společně* s tím, že testy mají před ostrým kódem náskok jen několik sekund.

Pokud budeme pracovat tímto způsobem, budeme denně psát desítky, měsíčně stovky a ročně tisíce testů. Pokud budeme pracovat tímto způsobem, tyto testy pokryjí prakticky veškerý ostrý kód. Čistý objem těchto testů, který může konkurovat objemu samotného ostrého kódu, může znamenat určité starosti pro ty, kteří jej mají udržovat.

## Mějte testy čisté

Před několika lety jsem byl požádán, abych vedl tým, jehož členové se otevřeně rozhodli, že jejich testovací kód *by se neměl* udržovat podle stejných norem kvality jako ostrý kód. Navzájem si tohle porušování jednotkových testů tolerovali. Heslem bylo „Rychlý a nečistý“. Nikdo nemusel hledat pro proměnné správné názvy, jejich testovací funkce nemusely být krátké a popisné. Jejich testovací kód nemusel být dobré navržený a promyšleně rozvržen. Dokud testovací kód fungoval a dokud pokrýval ostrý kód, stačilo to.

Někteří ze čtenářů by mohli s tímto rozhodnutím sympatizovat. Možná, že jste v dávné minulosti psali takové testy, jaké jsem napsal pro třídu `Timer`. Od psaní takového testu na jedno použití k na-psání sady automatizovaných jednotkových testů je to velký pokrok. Takže podobně jako tým, který jsem vedl, byste se mohli rozhodnout, že psaní nečistých testů je lepší než je nepsat vůbec.

Tomuto týmu nedošlo, že psaní nečistých testů je stejné, nebo snad i horší než nemít testy žádné. Problémem je, že s vývojem ostrého kódu je testy nutné aktualizovat. Čím je testovací kód spletitější, tím je pravděpodobnější, že strávíte více času na vložení nových testů do jejich sady, než kolik zabere psaní nového ostrého kódu. S modifikací ostrého kódu přestanou staré testy procházet a zmatek v testovacím kódu bude komplikovat vaše úsilí, aby testy opětovně prošly. Takže začnete na testy nahlížet jako na neustálé narůstající přítěž.

Náklady na údržbu testů rostly od jedné verze k druhé. Nakonec se z toho stal mezi vývojáři největší důvod jejich nespokojenosti. Když se jich manažeři ptali, proč jsou jejich odhadů tak vysoké, vývojáři uváděli jako příčinu testy. Nakonec byli nuceni odstranit je úplně.

Ale bez sady testů ztratili schopnost si ověřovat, že změny v jejich kódu aplikace pracují tak, jak očekávali. Bez testovací sady nemohli zabezpečit, aby změny v jedné části systému nenapáchaly škody v jeho ostatních částech. Takže počet chyb začal vzrůstat. S rostoucím počtem bezděčných chyb se vývojáři začali dalších změn obávat. Přestali s procistováním ostrého kódu, protože se báli, že změny by způsobily více škody než užitku. Jejich ostrý kód začal degenerovat. Nakonec neměli žádné testy, zato měli zmatený kód plný chyb, frustrované zákazníky a pocit, že jejich testovací úsili nesplnilo jejich očekávání.

Svým způsobem měli pravdu. Jejich testovací úsilí je opravdu *zklamalo*. Ale bylo to jejich rozhodnutí, že dovolili, aby testy byly zaneřáděné, a to byl pravý počátek jejich nezdaru. Kdyby byly jejich testy čisté, jejich testovací úsilí by je nezklamalo. Mohu to říci s velkou jistotou, protože jsem se zúčastnil práce v mnoha týmech (nebo jsem je i vedl), v nichž byly v otázce čistých jednotkových testů úspěšné.

Z této historky plyně jednoduché ponaučení: *Testovací kód je stejně důležitý, jako kód ostrý*. Nejde o druhoráděho občana. Vyžaduje přemýšlení, pečlivost při návrhu a pozornost. Je třeba udržovat ho ve stejně dobrém stavu jako ostrý kód.

## Testy otevírají další možnosti

Jestliže nebudete mít své testy bezchybné, přijdete o ně. A bez nich ztratíte vše, co udržuje váš ostrý kód flexibilní. Ano, čtete správně. Jsou to jednotkové testy, které udržují váš kód flexibilní, udržovatelný a opakovaně použitelný. Z jednoduchého důvodu. S testy se nemusíte bát provádět ve svém kódu změny! Bez nich je každá změna potenciální chybou. Nezáleží na flexibilitě vaší architektury, nezáleží na tom, jak je váš návrh rozvržen. Bez testů nebudeš ochotni provádět změny díky obavám, že zavedete těžko odhalitelné chyby.

Ale s *testy* tato obava prakticky zmizí. Čím lépe budete mít kód pokrytý testy, tím méně se budete obávat. Změny můžete provádět téměř beztrestně i v kódu, který není zdaleka prvotřídní a má spljit a neprůhledný návrh. Skutečně tuto architekturu a návrh můžete dále bez obav *vylepšovat!*

Takže s automatizovanou sadou jednotkových testů, která pokrývá ostrý kód, máte klíč k udržování svého návrhu a architektury tak bezchybné, nakolik je to možné. Testy odkrývají další možnosti, protože umožňují změnu.

Jestliže tedy nejsou vaše testy v pořádku, je vaše schopnost ke změnám omezená a začínáte ztrácet schopnost zlepšit strukturu kódu. Čím jsou vaše testy horší, tím horší bude váš kód. Pokud nakonec skončíte bez testů, ostrý kód zdegeneruje.

## Čisté testy

Co dělá testy čistými? Tři věci. Čitelnost, čitelnost a čitelnost. Možná, že je u jednotkových testů čitelnost dokonce důležitější než samotný ostrý kód. Co činí naše testy čitelnými? Totéž, co činí veškerý kód čitelným: jasnost, jednoduchost a schopnost se vyjádřit. V testech toho chcete hodně říci s tak málo výrazu, jak je to jen možné.

Podívejte se na kód z projektu FitNesse na výpisu 9.1. Není jednoduché těmto třem testům porozumět a určitě je možné je zlepšit. Zaprvé, v opakovaných voláních metod `addPage` a `assertSubString` je hrozné množství zdvojeného kódu [O5]. Co je ještě důležitější, tento kód je prostě zahlcen detaily, které překážejí expresivitě testu.

### Výpis 9.1. SerializedPageResponderTest.java

```
public void testGetPageHierarchyAsXml() 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 testGetPageHierachyAsXmlDoesntContainSymbolicLinks()
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 testDataAsHtml() 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);
}
```

Podívejte se například na volání metody `PathParser`. Transformuje řetězce do instancí typu `PagePath`, které používají metodu `crawler`. Tato transformace je v tomto testu zcela irrelevantní a pouze zatemňuje jeho smysl. Detaily, které obklopují vytváření metody `responder`, shromažďování a přety-povávání hodnót z metody `response`, to vše je rovněž jen šum. Pak je tam nešikovný způsob, jakým se vytváří požadovaná adresa URL z objektu `resource` a z nějakého dalšího argumentu. (Při psaní tohoto kódu jsem pomáhal, takže se cítím být oprávněný jej bez obalu kritizovat.)

Tento kód nebyl koneckonců navrhován proto, aby jej někdo četl. Chudák čtenář je zahlušen spoustou detailů, kterým musí porozumět, dříve než se pustí do jakýchkoliv smysluplných testů.

Podívejte se nyní na vylepšené testy ve výpisu 9.2. Tyto testy dělají přesně totéž, ale byly refaktorovány do mnohem čistší a vysvětlující podoby.

#### Výpis 9.2. SerializedPageResponderTest.java (po refaktorování)

```
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>>);
}
```

Ze struktury těchto testů je zřejmý vzor vytvoř-proveď-zkontroluj (BUILD-OPERATE-CHECK)<sup>2</sup>. Každý test je zřetelně rozdělen na tři části. První část generuje testovací data, druhá s těmito daty pracuje a třetí kontroluje, zda jsou získané výsledky správné.

Všimněte si, že většina detailů, které jsou jen na obtíž, byla odstraněna. Testy jdou přímo k věci a používají jen ta data a funkce, které doopravdy potřebují. Po přečtení těchto testů může kdokoliv rychle zjistit, co dělají, a to bez zavádějícího a ohromného množství detailů.

2. <http://fitnesse.org/FitNesse.AcceptanceTestPatterns>.

## Doménově specifický testovací jazyk

Testy z výpisu 9.2 ukazují techniku tvorby doménově specifického jazyka určeného pro vaše testy. Místo API, které programátoři používají pro ovládání systému, vytváříme sadu funkcí a nástrojů, jež tato API používají a se kterými se testy jednodušeji píšou i čtou. Tyto funkce a nástroje se stávají specializovanými částmi API, které testy využívají. Jsou testovacím *jazykem*, který používají programátoři, aby si usnadnili psaní svých vlastních testů a aby byly užitečné téměř již budou muset tyto testy číst později.

Tohle testovací API není primárně navrhováno pro tento účel, ale spíše se vyvíjí na základě pokračujícího refaktorování testovacího kódu, který je poznamenán matoucími detaily. Tak, jako jste mě viděli refaktorovat kód z výpisu 9.1 do podoby na výpisu 9.2, budou i disciplinovaní vývojáři refaktorovat svůj testovací kód do stručnější a expresivnější formy.

## Dvojí standard

V určitém smyslu dělal tým, o kterém jsem se zmínil na začátku této kapitoly, svou práci dobře. Kód, který je součástí testovacího API, má jiné technické standardy než ostrý kód. Musí být jednoduchý, stručný a expresivní, ale nemusí být tak efektivní jako ostrý kód. Nakonec funguje v testovacím prostředí, nikoliv v ostrém, a tato dvě prostředí mají velmi rozdílné požadavky.

Podívejte se na test z výpisu 9.3. Tento test jsem napsal jako součást prostředí řídicího systému, jehož prototyp jsem navrhoval. Aniž bychom zabíhali do detailů, můžeme říci, že tento test dohlíží na to, zda je v případě „příliš nízké teploty“ zapnuta signalizace nízké teploty, ohřívač či ventilátor.

### Výpis 9.3. EnvironmentControllerTest.java

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
}
```

Je zde samozřejmě mnoho detailů. Co například vlastně dělá funkce `tic`? Ale já bych vám spíše poradil, abyste se jimi při čtení těchto testů neznepokojovali. Bude lepší, když se budete soustředit na to, zda souhlasíte, že výsledný stav systému je konzistentní s podmínkou teploty, která je „příliš nízká“.

Všimněte si, že vaše oči musejí během čtení skákat dopředu a dozadu mezi názvem sledovaného stavu a *smyslem* tohoto stavu. Podíváte se na metodu `heaterState` a pak vaše oči skložou vlevo na `assertTrue`. Podíváte se na metodu `coolerState` a vaše oči musejí přejít vlevo na `assertFalse`. To je únavné a nespolehlivé. To čitelnost testů komplikuje.

Čitelnost tohoto testu jsem značně zlepšil tím, že jsem jej transformoval do kódu ve výpisu 9.4.

**Výpis 9.4.** EnvironmentControllerTest.java (po refaktorování)

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}
```

Detail funkce tic jsem samozřejmě ukryl tím, že jsem vytvořil funkci `wayTooCold()`. Ale za povšimnutí stojí zvláštní řetězec v metodě `assertEquals`. Velké písmeno znamená „zapnuto“, malé písmeno znamená „vypnuto“ a pořadí písmen je vždy následující: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.

I když jsme málem porušili pravidlo ohledně skrytého překládání jmen<sup>3</sup>, v tomto případě se to zdá být vhodné. Všimněte si, že jak jednou znáte význam, mohou vaše oči řetězec přelétnout a můžete výsledky rychle interpretovat. Čtení testu pak pro vás bude téměř radostný akt. Podívejte se na výpis 9.5 a přesvědčte se, jak je jednoduché jim porozumět.

**Výpis 9.5.** EnvironmentControllerTest.java (bigger selection)

```
@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());
}
```

Výpis 9.6 zobrazuje funkci `getState()`. Všimněte si, že to není příliš efektivní kód. Abych tuto vlastnost zlepšil, měl jsem pravděpodobně používat `StringBuffer`.

**Výpis 9.6.** MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
```

3. „Vyhnete se skrytému překládání jmen“ na straně 46.

```

state += hiTempAlarm ? "H" : "h";
state += loTempAlarm ? "L" : "l";
return state;
}

```

Proměnné typu `StringBuffer` jsou poněkud nehezké. I v ostrém kódu se jim budu vyhýbat, pokud cena za tento krok nebude příliš velká. Lze říci, že náklady na kód z výpisu 9.6 jsou velmi malé. Tato aplikace je však na první pohled nějakým vnořeným systémem pracujícím v reálném čase. Ale *testovací* prostředí by nemělo být nijak omezováno.

To je podstata dvojího standardu. Existují věci, které byste nikdy neměli dělat v prostředí ostrého kódu, ale jež jsou zcela využitelné v testovacím prostředí. Obvykle se jedná o využití paměti nebo efektivity procesoru. Ale *nikdy* se nejedná o otázku čistoty.

## Jedna aserce na jeden test

Existuje názor<sup>4</sup>, který říká, že každá testovací funkce v testech JUnit by měla mít právě jeden příkaz `assert`. Toto pravidlo se může zdát drakonické, ale z výpisu 9.5 je patrná jeho výhoda. Takové testy vedou k jednomu závěru, který je snadno a rychle srozumitelný.

Ale co s výpisem 9.2? Zdá se být nevhodné, že můžeme nějak jednoduše sloučit Aserci prověřující, že výstup je ve formátu XML, a skutečnost, že obsahuje určité podřetězce. Ale tento test můžeme rozdělit na dva samostatné testy, z nichž bude mít každý svou vlastní aserci, jak ukazuje výpis 9.7.

### Výpis 9.7. SerializedPageResponderTest.java (Single Assert)

```

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>"
    );
}

```

Všimněte si, že jsem změnil názvy funkcí, abych mohl používat běžnou konvenci `given-when-then`<sup>5</sup> (dáno-když-pak). To dokonce čitelnost testů zlepšuje. Tento způsob rozdělení testů vede bohužel ke zdvojení kódu.

To můžeme odstranit použitím vzoru šablonová metoda (TEMPLATE METHOD)<sup>6</sup>, když vložíme části kódu `given/then` do základní třídy a části kódu `then` do odvozených tříd. Mohli bychom také vytvořit třídu testů zcela samostatně a vložit části `given` a `when` do funkce `@Before` a části `then` do

- 
4. Viz záZNAM v blogu Dava Astela:  
<http://www.artima.com/weblogs/viewpost.jsp?thread=35578>.
  5. [RSpec].
  6. [GOF].

každé z funkcí @Test. Ale zdá se, že tento mechanismus je tak trochu kanón na komára. Nakonec bych dal přece jen přednost několikanásobným asercím z výpisu 9.2.

Myslím si, že pravidlo jedné aserce je dobré vodítko<sup>7</sup>. Obvykle se pokouší vytvořit doménově specifický testovací jazyk, který tohle pravidlo podporuje, jak ukazuje výpis 9.5. Ale nebojím se vkládat do testu více asercí. To nejlepší, co zřejmě můžeme říci, je, že bychom měli v testech počet asercí minimalizovat.

## Jedna myšlenka pro jeden test

Možná, že lepším pravidlem je, budeme-li testovat v každé testovací funkci jen jednu myšlenku. Nechceme mít dlouhé testovací funkce, které testují různé věci jednu za druhou. Výpis 9.8 je ukázka takového testu. Tento test by měl být rozdělen na tři nezávislé testy, protože testuje tři nezávislé věci. Jejich sloučení do jedné funkce nutí čtenáře, aby určoval, proč je která sekce tam a tam a co vlastně testuje.

### Výpis 9.8. Test testující tři nezávislé věci

```
/**
 * Různé 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());
}
```

Tyto tři testovací funkce by pravděpodobně měly vypadat takto:

- ◆ *Dáno (given)*: poslední den v měsíci s 31 dny (například květen):
  1. *Když (when)* přidáte takový měsíc, aby byl jeho poslední den co do počtu třicátý (jako je červen), *pak (then)* by mělo být datum třicátého daného měsíce a ne jednatřicátého.
  2. *Když (when)* k tomuto datu přičtete dva měsíce tak, aby poslední měsíc měl 31 dnů, *pak (then)* by poslední datum mělo být jednatřicátého.
- ◆ *Dáno*: poslední den v měsíci, který má 30 dní (například červen):
  1. *Když (when)* přičtete jeden měsíc, který má 31 dní, *pak (then)* by výsledné datum mělo být třicátého a ne jednatřicátého.

7. „Držte se kódu!“

S takto definovanými podmínkami se můžete přesvědčit, že se mezi různorodými testy skrývá jedno obecné pravidlo. Když zvýšíte číslo měsíce o jedničku, nemůže být datum větší, než je poslední den v měsíci. To znamená, že přičtením jednotky k osmadvacátému únoru dostaneme osmadvacátý březen. *Tento* test nám chybí a bylo by dobré jej napsat.

Takže problém zde nespočívá ve více asercích v každé sekci výpisu 9.8 a není to příčinou problému. Problémem je spíše skutečnost, že existuje více testovaných myšlenek. Takže pravděpodobně nejlepším pravidlem je minimalizovat počet asercí na myšlenku a v rámci testovací funkce testovat jen jeden z nich.

## F.I.R.S.T.<sup>8</sup>

Čisté testy sledují pět dalších pravidel, ze kterých se výše uvedená zkratka skládá:

**Fast** (rychlý) – Testy by měly být rychlé. Měly by běžet rychle. Pokud běží pomalu, nebudeste je chtít spouštět příliš často. Pokud je nebudeste spouštět často, nenaleznete včas problémy, abyste je mohli řešit jednoduchým způsobem. Budete se obávat čistění kódu. Ten nakonec začne degenerovat.

**Independent** (nezávislý) – Testy by na sobě neměly záviset. Jeden test by neměl nastavovat podmínky pro test následující. Každý test by se měl spouštět nezávisle a měli byste být schopni je spouštět v libovolném pořadí. Pokud testy na sobě závisejí, selhání prvního způsobí kaskádu selhání těch ostatních, což způsobí potíže při diagnostice a skryje chyby, které by následné testy odhalily.

**Repeatable** (opakovatelný) – Testy by měly být v jakémkoliv prostředí opakovatelné. Měli byste být schopni je spouštět v ostrém prostředí, v prostředí QA nebo na svém notebooku během cesty domů vlakem, když jste odpojeni od sítě. Jestliže testy nelze zopakovat v libovolném prostředí, máte vždy výmluvu, proč neprošly. Nebudeste je moci spustit, ani když budete mít požadované prostředí právě k dispozici.

**Self-Validating** (samovyhodnocující) – Testy by měly mít výstup ve formě logické hodnoty. Budě prošly, nebo neprošly. Neměli byste být nuteni čist protokolovací soubor nebo ručně porovnávat dva různé textové soubory, abyste se zjistili, zda testy prošly. Pokud se testy nedokážou vyhodnotit samy, může jejich selhání být věcí subjektivního názoru a jejich spouštění může následně vyžadovat pracné ruční vyhodnocování.

**Timely** (aktuální) – Testy by se měly psát aktuálně. Jednotkové testy by se měly napsat *těsně před* ostrým kódem, který je bude spouštět. Pokud napíšete testy po ostrém kódu, můžete zjistit, že je obtížné ostrý kód testovat. Můžete dospět k závěru, že některá část ostrého kódu je pro testování prostě obtížná. Nebo navrhnete kód, který testovat nelze.

## Závěr

Tohle téma jsme opravdu jen lehce nakousli. Jsem toho názoru, že na téma *čistých testů* by mohla být napsaná celá kniha. Testy jsou pro zdraví projektu tak důležité, jako je samotný ostrý kód. Možná, že jsou dokonce ještě důležitější, protože v sobě zahrnují a podporují flexibilitu, správu a opakovatelnou použitelnost ostrého kódu. Takže udržujte své testy stále čisté. Snažte se, aby byly expresivní

8. Studijní materiály firmy Object Mentor.

a stručné. Vytvořte testování rozhraní API, které funguje jako doménově specifický jazyk, jenž vám bude při psaní testů užitečný.

Pokud necháte **testy** zdegenerovat, váš kód zdegeneruje také. Udržujte je čisté.

## Použitá literatura

[RSpec]: *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesøy, David Chelimsky, Pragmatic Bookshelf, 2008.

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

# KAPITOLA 10

## Třídy

*Jeff Langr*

### V této kapitole najdete:

- ◆ Organizace třídy
- ◆ Třídy by měly být malé!
- ◆ Organizace podporující změny
- ◆ Použitá literatura



Dosud jsme se v této knize věnovali pouze tomu, jak dobře psát bloky kódu nebo jeho jednotlivé řádky. Ponořili jsme se do správné kompozice funkcí a jejich vzájemných vztahů. Ale přes veškerou pozornost k expresivitě příkazů kódu a funkcí nebude nás kód čistý, pokud se nebudeme věnovat vyšším úrovním jeho organizace. Pojďme si říci něco o čistých třídách.

## Organizace třídy

Podle konvence pro Javu by měla třída začínat seznamem proměnných. Na prvním místě by měly být veřejné statické konstanty, pokud nějaké obsahuje. Po nich by měly následovat statické proměnné (datové složky) a po nich soukromé instanční proměnné. Pro použití veřejných proměnných většinou neexistují dobré důvody.

Po seznamu proměnných by měly následovat veřejné funkce. Bezprostředně za nimi nejlépe umístíme soukromé nástroje, které tyto veřejné funkce volají. To je v souladu s metodou sestupu a přispívá to k tomu, abychom mohli číst program tak, jak čteme novinový článek.

### Zapouzdření

Dáváme přednost tomu, aby naše proměnné a pomocné funkce byly soukromé, ale nejsme v tomto směru fanatičtí. Někdy je zapotřebí, aby byla proměnná nebo pomocná funkce chráněná a takto byla přístupná nástrojům pro testování. Z našeho úhlu pohledu jsou testy zcela zásadní. Pokud potřebujeme test ze stejněho balíčku volat funkci nebo přistupovat k proměnné, vytvoříme ji jako chráněnou nebo dostupnou v rámci balíčku. V každém případě se je však především pokusíme zachovat jako soukromé. Zrušit zapouzdření je vždy až tou poslední variantou.

## Třídy by měly být malé!

Prvním pravidlem pro třídy je, že by měly být malé. Druhým pravidlem je, že by měly být ještě menší. Nikoliv, nehdálám zde opakovat doslovné znění textu z kapitoly *Funkce*. Ale podobně jako u funkcí je u tříd primárním pravidlem, aby byly menší. Podobně jako u funkcí bude bezprostředně následovat otázka „jak malá?“ Rozsah funkcí měříme podle počtu fyzických řádků. U tříd používáme jinou metodu. Sčítáme *odpovědnosti*<sup>1</sup>. Výpis 10.1 ukazuje třídu SuperDashboard, která nabízí přibližně 70 veřejných metod. Většina vývojářů by souhlasila, že na velikost třídy to je poněkud příliš. Jiní by ji naopak mohli považovat za „dobrou“.

#### Výpis 10.1. Příliš mnoho odpovědností

```
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)
```

1. [RDD].

```
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()
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 actionPerformed, 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 setAllowDragging(boolean allowDragging)
public boolean allowDragging()
public boolean isCustomizing()
public void setTitle(String title)
public IdeMenuBar getIdeMenuBar()
public void showHelper(MetaObject metaObject, String propertyName)
// ... následuje mnoho dalších neveřejných metod ...
}

```

Ale co kdyby třída SuperDashboard obsahovala pouze metody uvedené ve výpisu 10.2?

#### Výpis 10.2. Je dostatečně krátká?

```

public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}

```

Pět metod není zas tak mnoho, že? V tomto případě ano, protože i navzdory malému počtu metod má třída SuperDashboard příliš mnoho *odpovědností*.

Název třídy by měl popisovat, které odpovědnosti funkce vykonává. Ve skutečnosti je její název pravděpodobně prvním krokem, který určuje její velikost. Pokud u třídy nelze odvodit její stručný název, pak je pravděpodobně příliš velká. Čím je název třídy mnohoznačnější, tím spíše bude mít příliš mnoho odpovědností. Například názvy, obsahující mnohoznačná jména jako Processor, Manager nebo Super často naznačují nešťastnou agregaci odpovědností.

Měli bychom být rovněž schopni napsat krátký popis třídy pomocí pětadvaceti slov, aniž bychom museli použít slova, jako je „když“, „a“, „nebo“ či „ale“. Jak bychom mohli popsat třídu SuperDashboard? Tato třída umožňuje přístup ke komponentě, která měla naposledy fokus a rovněž nám umožňuje zjistit verzi a zabudovaná čísla. První slovíčko na začátku vedlejší věty „a“ naznačuje, že třída SuperDashboard má příliš mnoho odpovědností.

## Princip jediné odpovědnosti

Princip jediné odpovědnosti (SRP – Single Responsibility Principle)<sup>2</sup> říká, že třída nebo modul by měly mít jeden a jen jeden důvod ke změně. Tento princip nám dává jak definici odpovědnosti, tak i vodítko pro její velikost. Třídy by měly mít jedinou odpovědnost – jediný důvod ke změně.

Ve výpisu 10.2 má zdánlivě malá třída SuperDashboard hned dva důvody ke změně. Za prvé zjišťuje informaci o verzi, která by zřejmě měla být aktualizovaná pokaždé, když je software odesílán zákazníkovi. Za druhé spravuje komponenty Java Swing (varianta okna GUI vysoké úrovně, odvozená od

2. Mnohem více si o tomto principu můžete přečíst v [PPP].

JFrame). Není pochyb, že bychom chtěli aktualizovat číslo verze, jestliže na kódu komponent Swing cokoliv změníme, ale nemusí to nutně platit naopak: informaci o verzi můžeme chtít změnit na základě změn nějakého jiného kódu v systému.

Pokus o identifikaci odpovědnosti (důvodů ke změnám) nám často pomáhá rozpoznat a vytvářet v kódu lepší abstrakce. Ze třídy SuperDashboard můžeme jednoduchým způsobem extrahat všechny tři metody, zabývající se informacemi o verzi, do samostatné třídy Version. (Viz výpis 10.3.) Tato třída je konstrukcí, u níž je velmi pravděpodobné opakované využití i v jiných aplikacích!

#### Výpis 10.3. Třída s jednou odpovědností

```
public class Version {  
    public int getMajorVersionNumber()  
    public int getMinorVersionNumber()  
    public int getBuildNumber()  
}
```

Princip jediné odpovědnosti je v návrhu objektově orientovaného programování jeden z důležitějších pojmu. Je to také jeden z jednoduchých pojmu, kterým lze snadno porozumět a řídit se jimi. Ale kupodivu je tento princip při návrhu tříd jedním z nejčastěji porušovaných pravidel. Pravidelně narázíme na třídy, které dělají příliš mnoho věcí. Proč?

Funkční software a čistý software jsou dvě značně rozdílné činnosti. Pozornost je u většiny z nás omezená, díky čemu se soustředíme především na to, aby software fungoval, a teprve pak na jeho organizaci a čistotu. To je celkem pochopitelné. Starat se o oddělování úkolů je stejně tak důležité jak v naši programovací činnosti, tak i v našich programech.

Problémem je, že si mnoho z nás myslí, že když nám funguje program, máme vše za sebou. Neumíme se přepnout na další záležitost, kterou je organizace a čistota kódu. Přecházíme na řešení dalšího problému, místo abychom se vrátili zpět a rozdělili přecpanou třídu do oddělených jednotek se samostatnými odpovědnostmi.

Mnoho vývojářů se také obává, že velké množství malých jednoúčelových tříd bude mít za důsledek obtížnější porozumění celku. Mají obavu, že budou muset přecházet z třídy na třídu, budou-li chtít stanovit, jak dát dohromady větší celek.

Avšak systém s mnoha třídami nemá více proměnlivých částí než systém s velkými třídami. Studovat systém s několika velkými třídami nebude o nic jednodušší, než systém s malými třídami. Takže otázka zní: chcete mít své náradí uloženo v příslušných schránkách s malými šuplíky, které by obsahovaly pečlivě definované a dobře popsané komponenty? Nebo jich chcete jen několik a všechno do nich prostě naházet?

Každý velký systém bude obsahovat velké množství logiky a komplikovanosti. Při správě takto komplikovaného systému je primárním cílem *zorganizovat* vše tak, aby vývojář kdykoliv věděl, kde najde vše potřebné, a aby potřeboval znát jen tu část, která bude ovlivněna. Naproti tomu systém s velkými víceúčelovými třídami nám bude vždy překážet v tom smyslu, že nás bude nutit procházet spoustu věcí, které právě ted' nepotřebujeme.

Abych zdůraznil ještě jednou výše uvedené argumenty: chceme, aby se naše systémy skládaly z mnoha menších tříd a ne z několika velkých. Každá malá třída v sobě zahrnuje jednu odpovědnost, má jen jeden důvod ke změně a spolupracuje s několika dalšími, abychom dosáhli požadované funknosti systému.

## Soudržnost

Třídy by měly mít malý počet instančních proměnných. Každá metoda dané třídy by měla pracovat s jednou nebo s několika proměnnými. Obecně platí, že čím větší je počet proměnných, se kterými metoda pracuje, tím je větší soudržnost s její třídou. Třída, ve které každá metoda používá každou proměnnou, je maximálně soudržná.

Obecně nelze vytváření takto maximálně soudržné třídy doporučit a není to ani možné. Na druhé straně bychom chtěli, aby soudržnost byla vysoká. Je-li vysoká, znamená to, že metody a proměnné této třídy na sobě závisejí a tvoří dohromady jeden logický celek.

Podívejte se na implementaci třídy Stack z výpisu 10.4. Jedná se o velmi soudržnou třídu. Ze tří metod, které obsahuje, jen jedna metoda, size(), nepoužívá obě proměnné.

### Výpis 10.4. Soudržná třída Stack.java

```
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;
    }
}
```

Strategie mít malé funkce a krátký seznam parametrů může někdy vést k vyššímu počtu instančních proměnných, jež používají jen některé třídy. Pokud k tomu dojde, téměř vždy to znamená, že existuje alespoň jedna další třída, která se snaží z té velké nějak vymanit. Měli byste se pokusit rozdělit proměnné i metody do dvou nebo více tříd tak, aby nové třídy byly soudržnější.

## Soudržnost vede k mnoho malým třídám

Samotný akt dělení velkých funkcí na menší způsobuje nárůst počtu tříd. Podívejte se na rozsáhlou funkci s deklaracemi mnoha proměnných. Dejme tomu, že chcete extrahovat jednu její menší část do samostatné funkce. Avšak kód, který chcete extrahovat, používá čtyři proměnné, jež jsou v této funkci deklarované. Musíte předat všechny tyto čtyři proměnné jako argumenty do nové funkce?

Naprosto ne! Kdybychom převedli tyto čtyři proměnné na instanční proměnné třídy, pak bychom mohli extrahovat kód bez jakéhokoliv předávání proměnných. Bylo by jednoduché tuto funkci rozdělit na několik menších částí.

Bohužel to také znamená, že naše třídy ztratí soudržnost, protože shromažďují více a více instančních proměnných, které existují pouze z toho důvodu, aby je mohlo sdílet více funkcí. Ale počkejte! Pokud existuje několik nových funkcí, které potřebují sdílet určité proměnné, nedělá to z nich třídu samu o sobě? Samozřejmě že ano. Když třídy ztrácejí soudržnost, rozdělte je!

Rozdělení velkých funkcí do několika menších vás někdy může vést také k vytvoření několika menších tříd. Pro nás program to znamená mnohem lepší organizaci a průhlednější strukturu.

Zkusme se podívat na jeden starší příklad z vynikající knihy Donalda Knutha „Literate Programming“<sup>3</sup> jako na ukázku toho, co mám na mysli. Výpis 10.5 ukazuje Knuthův program PrintPrimes po převodu do jazyka Java. Abychom byli vůči Knuthovi poctiví, tento program není v té podobě, jak jej napsal, ale je to výstup z jeho webového nástroje. Použil jsem jej, protože je to dobrý start pro rozdělení velké funkce na mnoho menších funkcí a tříd.

#### Výpis 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) {
                    ORD = ORD + 1;
                    SQUARE = P[ORD] * P[ORD];
                    MULT[ORD - 1] = J;
                }
                N = 2;
```

3. [Knuth92].

```
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("The First " + M +
        " Prime Numbers --- Page " + 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;
}
}
}
}
```

Tento program, napsaný jako jedna funkce, je chaotický. Má značně odsazenou strukturu, přemíru divných proměnných a těsně svázanou strukturu. Přesněji řečeno, tato velká funkce by měla být rozdělena na několik menších.

Výpis 10.6 až 10.8 ukazují výsledky rozdělení kódu z výpisu 10.5 na menší třídy a funkce s tím, že pro třídy, funkce a proměnné byly zvoleny smysluplnější názvy.

#### Výpis 10.6. PrimePrinter.java (po refaktorování)

```
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,
```

```
        "The First " + NUMBER_OF_PRIMES +
        " Prime Numbers");
    tablePrinter.print(primes);
}
}
```

**Výpis 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 + " --- Page " + pageNumber);
    printStream.println("");
}
public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
}
}
```

**Výpis 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;
}
}

```

První věcí, které si můžete všimnout, je, že je program mnohem delší. Z délky jedné stránky se rozrostl na téměř tři stránky. Máme pro to několik důvodů. Za prvé, refaktorovaný program používá delší a popisnější názvy proměnných. Za druhé, program používá takové deklarace funkcí a tříd, které dokážou kód komentovat. Za třetí, používá mezery a formátovací postupy tak, aby byl program čitelnější.

Všimněte si, jak byl program rozdělen na tři hlavní odpovědnosti. Hlavní program je obsažen v samotné třídě `PrimePrinter`. Jeho odpovědností je zpracovat prostředí, ve kterém program běží. To bude podléhat změnám, pokud bude podléhat změnám volající metoda. Bude-li například tento program převeden na službu SOAP, bude právě tato třída převodem ovlivněna.

Třída `RowColumnPagePrinter` ví vše o tom, jak na stránkách s daným počtem řádků a sloupců formátovat seznam čísel. Pokud bude nutné výstupní formátování změnit, povede to ke změnám v této třídě.

Třída `PrimeGenerator` umí generovat seznam prvočísel. Všimněte si, že není záměrem, aby vznikla její instance. Třída je pouze užitečnou oblastí, ve které je možné deklarovat proměnné a mít je ukryté. Třída bude podléhat změnám, pokud se změní algoritmus pro výpočet prvočísel.

Nejde o novou verzi! Nezačali jsme program psát znova a na zelené louce. Pokud se podíváte na oba programy trochu podrobněji, uvidíte, že oba používají pro svou práci stejný algoritmus a stejné mechanismy.

Změna začala napsáním sady testů, které ověřují přesné chování prvního programu. Poté jsme udělali postupně nesčetné malé změny a po každé z nich byl program spuštěn, abychom se ujistili, že se funkčnost nezměnila. Pomocí malých postupných kroků byl první program vyčistěn a transformován na jiný.

## Organizace podporující změny

Pro většinu systémů platí, že se v nich neustále provádějí změny. Každá z nich nás vystavuje riziku, že ostatní části systému nebudou fungovat tak, jak očekáváme. V čistém systému si zorganizujeme třídy tak, abychom rizika, vyplývající ze změn, omezili.

Třída `Sql` z výpisu 10.9 generuje správné tvary řetězců SQL, pokud obdrží vhodná metadata. Tato třída je ve stadiu rozpracovanosti a jako taková prozatím nepodporuje funkčnost SQL, jako je například příkaz `update`. Když nastane vhodná chvíle, aby třída `Sql` podporovala příkaz `update`, budeme muset tuto třídu „otevřít“ a provést modifikace. Problém s jejím otevřením je určité riziko. Jakákoliv její modifikace v sobě skrývá nebezpečí porušení jiného kódu třídy a je zapotřebí ji důkladně otestovat.

**Výpis 10.9.** Modifikovanou třídu je třeba otevřít

```
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)
}
```

Při přidávání nového typu příkazu musíme třídu `Sql` modifikovat. Musíme ji modifikovat také v případě, že měníme detaily nějakého jednoduchého příkazu – například pokud potřebujeme modifikovat činnost příkazu `select` a takto podporovat vnořené příkazy `select`. Tyto dva důvody ke změnám znamenají, že třída `Sql` porušuje pravidlo jedné odpovědnosti.

Všimněme si tohoto porušení pravidla jedné odpovědnosti z hlediska organizace. Kostra metody `Sql` ukazuje, že existují soukromé metody, jako třeba `selectWithCriteria`, které zřejmě mají vztah jen k příkazům `select`.

Funkčnost soukromých metod, která se vztahuje jen na malou podmnožinu prvků třídy, může být velmi užitečnou heuristikou pro určení potenciálních částí kódu, jež chceme vylepšit. Avšak první pobídka pro nějakou akci by měl být systém sám. Pokud zastáváme názor, že logika třídy `Sql` je hotová, pak se nemusíme bát odpovědnosti oddělit. Pokud nebudeme v nejbližší budoucnosti potřebovat funkcionality příkazu `update`, mohli bychom ponechat třídu `sql` tak, jak je. Ale jakmile bude třídu otevírat, měli bychom se nad opravou našeho návrhu zamyslet.

Co kdybychom se přiklonili k řešení z výpisu 10.10? Každá metoda veřejného rozhraní, definovaná v předchozí třídě `Sql` ve výpisu 10.9 byla podrobena refaktorování a je odvozená od třídy `Sql`. Všimněte si, že soukromé metody, jako je `valuesList`, byly přemístěny tam, kde jsou zapotřebí. Všeobecná soukromá funkčnost je izolována do dvou pomocných tříd `Where` a `ColumnList`.

#### Výpis 10.10. Sada uzavřených tříd

```
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()  
    }  
}
```

Kód v těchto třídách se velmi zjednoduší. Čas, nutný pro porozumění kterékoliv z těchto tříd, se sníží téměř na nulu. Riziko, že by jedna z funkcí mohla narušit chod jiné funkce, se snižuje na minimum. Z pohledu testování bude ověřování všech částí logiky tohoto řešení jednodušší, protože třídy jsou navzájem izolované.

Stejně tak je důležité, že když budeme přidávat příkazy update, nebudeme muset měnit žádnou z existujících tříd! Logiku příkazů update napišeme do nové třídy `UpdateSql`, která je podtřídou `Sql`. Nebude poškozen žádný jiný kód tohoto systému.

Restrukturalizovaná logika třídy `Sql` reprezentuje to nejlepší, co můžeme mít. Vyhovuje pravidlu jediné odpovědnosti. Vyhovuje i jinému klíčovému principu objektově orientovaného programování, známému jako princip otevřenosti a uzavřenosti (OCP – Open-Closed Principle)<sup>4</sup>: Třídy by měly být otevřené pro rozšiřování, ale uzavřené pro modifikace. Naše restrukturalizovaná třída `Sql` je otevřená, aby umožnila zavedení nové funkčnosti vytvářením podtříd, ale tyto změny můžeme provádět, i když jsou všechny ostatní třídy uzavřené. Vložíme tam jednoduše třídu `UpdateSql`. Chceme své systémy strukturovat tak, abychom je při jejich aktualizaci nebo při přidávání nových funkčností zaneřádili co nejméně. V ideálním systému zavedeme nové funkčnosti rozšiřováním systému, ne modifikacemi jejich existujícího kódu.

## Izolování od změn

Budou se měnit potřeby, s nimi se bude měnit i kód. V základním kursu objektově orientovaného programování jsme se naučili, že existují konkrétní třídy, které obsahují implementační detaily (kód) a abstraktní třídy, jež reprezentují pouze pojmy. U klientských tříd, které závisejí na konkrétních detailech, existuje určité riziko, že se tyto detaily změní. Abychom pomohli izolovat dopad těchto detailů, můžeme pro tento účel zavést rozhraní a abstraktní třídy.

Závislosti na konkrétních detailech znamenají pro nás testovací systém problémy. Pokud vytváříme třídu `Portfolio`, která závisí na externím modulu `TokyoStockExchange` z API, a chceme získat hodnotu portfolia, jsou naše testovací případy ovlivněny různými výsledky takového prohledávání. Je těžké napsat test tam, kde můžeme každých pět minut obdržet jinou odpověď!

Místo abychom vytvářeli třídu `Portfolio`, která závisí přímo na `TokyoStockExchange`, vytvoříme rozhraní `StockExchange`, které deklaruje jedinou metodu:

```
public interface StockExchange {
    Money currentPrice(String symbol);
}
```

Vytvoříme modul `TokyoStockExchange`, aby toto rozhraní implementoval. postaráme se také o to, aby konstruktor třídy `Portfolio` převzal odkaz na `StockExchange` jako argument:

```
public Portfolio {
    private StockExchange exchange;
    public Portfolio(StockExchange exchange) {
        this.exchange = exchange;
    }
    // ...
}
```

4. [PPP].

Nyní může nás test vytvořit testovatelnou implementaci rozhraní StockExchange, které emuluje modul TokyoStockExchange. Tato implementace způsobí, že aktuální hodnota pro libovolný symbol, který použijeme během testování, bude neměnná. Pokud nás test zobrazí prodej pěti akcií firmy Microsoft našeho portfolia, napišeme testovací implementaci tak, aby vždy vracela sto dolarů na akci firmy Microsoft. Implementace testovacího rozhraní StockExchange se zredukuje na jednoduché prohledávání tabulky. Můžeme pak napsat test, který jako celkovou hodnotu portfolia očekává sumu pěti set dolarů.

```
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());  
    }  
}
```

Pokud bude systém v dostatečné míře zbaven vazeb a bude možné jej takto testovat, bude i pružnější a bude podporovat opakované využívání kódu. Odstranění vazeb znamená, že se jednotlivé prvky systému lépe izolují jak od změn, tak i navzájem. Tato izolace usnadňuje pochopení každého prvku systému.

Minimalizací počtu vazeb se naše třídy blíží jinému návrhovému pravidlu, známému jako princip inverze závislostí (DIP – Dependency Inversion Principle)<sup>5</sup>. Toto pravidlo v podstatě říká, že by třídy měly záviset na abstrakcích a ne na konkrétních detailech.

Místo abychom záviseli na implementaci detailů třídy TokyoStockExchange, závisí nyní třída Portfolio na rozhraní StockExchange. Tohle rozhraní reprezentuje abstraktní pojem dotazu na aktuální cenu symbolu. Tato abstrakce izoluje veškeré specifické detaily, spojené se získáváním ceny, včetně informace, odkud tato cena pochází.

## Použitá literatura

[RDD]: *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

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

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

5. [PPP].



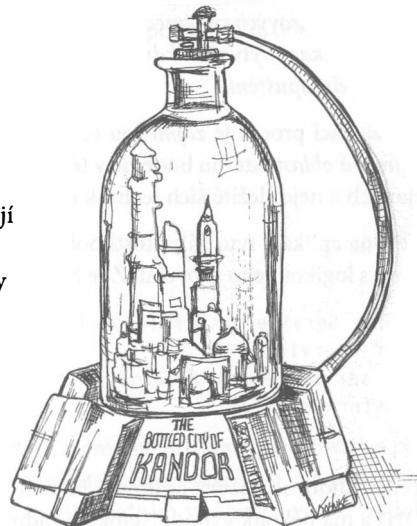
# KAPITOLA 11

## Systémy

*Dr. Kevin Dean Wampler*

### V této kapitole najdete:

- ◆ Jak byste postavili město?
- ◆ Oddělujte tvorbu systému od jeho používání
- ◆ Škálování
- ◆ Zprostředkovatel v Javě
- ◆ Čisté rámce Java AOP
- ◆ Aspekty AspectJ
- ◆ Testování systémové architektury
- ◆ Optimalizujte rozhodování
- ◆ Používejte rozumně standardy, pokud přinášejí prokazatelnou hodnotu
- ◆ Systémy potřebují doménově specifické jazyky
- ◆ Závěr
- ◆ Použitá literatura



*„Složitost zabíjí. Vysává z vývojářů život, ztěžuje plánování produktů, jejich sestavování a testování.“*

Ray Ozzie, CTO, Microsoft Corporation

## Jak byste postavili město?

Jste schopni zvládnout ty detaily sami? Pravděpodobně ne. Řízení i existujícího města je pro jednoho člověka příliš těžkým oříškem. Přesto města fungují (většinou). Jsou v provozu, protože mají týmy lidí, kteří řídí určité jeho části, jako jsou vodovodní systémy, energetika, provoz, soudní aparát, tvorba předpisů a podobně. Někteří z těchto lidí zodpovídají za celek, jiní se soustředí na detaily.

Města fungují také proto, že se zde vyvinuly náležité úrovně abstrakce a modularity, které umožňují jednotlivcům a jejich „komponentám“ pracovat efektivně, aniž by museli rozumět celku.

Ačkoliv jsou softwarové týmy často organizovány na podobném základě, systémy, se kterými pracují, mírají jiné rozdělení zájmových oblastí a úrovně abstrakce. Na nízké úrovni abstrakce nám k dosahování těchto cílů pomáhá čistý kód. V této kapitole se budeme zabývat tématem, jak zachovat čistý kód na vyšších úrovních abstrakce – na úrovni *systému*.

## Oddělujte tvorbu systému od jeho používání

Především si uvědomte, že *vytváření* je zcela něco jiného, než je *používání*. V době, kdy píšu tyto řádky, vidím ze svého okna v Chicagu stavbu nového hotelu. Dnes je to holá betonová krabice se stavebním jeřábem a výtahem připevněným k jejímu vnějšímu povrchu. Všichni, co tam pracují, nosí ochranné přilby a pracovní obleky. Hotel bude dokončen přibližně do jednoho roku. Jeřáb i výtah budou pryč. Budova bude čistá, pokrytá skleněnými zdmi s okny a bude mít atraktivní náter. Také lidé, kteří v ní budou pracovat a bydlet, budou vypadat zcela jinak.

*V softwarových systémech bychom měli oddělovat proces zahájení výstavby, kdy se jednotlivé objekty aplikace vytváří a kdy se navzájem propojují závislosti, od logiky jeho provozu, který následuje po jeho spuštění.*

Zahajovací proces je *zájmovou oblastí*, se kterou se musí vypořádat každá aplikace. Je to ta první *zájmová oblast*, kterou budeme v této kapitole zkoumat. *Oddělení zájmových oblastí* je jednou z nejstarších a nejdůležitějších technik návrhů v našem oboru.

Většina aplikací tyto záležitosti bohužel neodděluje. Kód na začátku procesu je ad hoc a je promíchan s logikou jeho provozu. Zde máme typický příklad:

```
public Service getService() {
    if (service == null)
        service = new MyServiceImpl(...); // Je tato výchozí hodnota dostatečná
    return service; // pro většinu případů?
}
```

Tento idiom se nazývá odložená inicializace/vyhodnocení (LAZY INITIALIZATION/EVALUATION) a má několik výhod. Nemáme žádné režijní náklady na konstrukci objektů, dokud je opravdu nepoužijeme, což může zkrátit celkový startovací čas. Rovněž máme jistotu, že se nám nikdy nebude vracet hodnota `null`. Je zde však pevně zakódovaná závislost na metodě `MyServiceImpl` a na všem,

co vyžaduje její konstruktor (který jsem vynechal). Bez vyřešení těchto závislostí nebude možné kód přeložit, i kdybychom tento objekt během provozu nikdy nepotřebovali!

Testování může být problémem. Je-li `MyServiceImpl` objektem zásadního významu, budeme si muset zajistit, aby byl této položce služby přiřazen náležitý napodobující objekt<sup>1</sup> ještě před voláním této metody během jednotkového testování. Protože logika konstrukce je promíchána s logikou běžného provozního procesu, měli bychom otestovat všechny větve (například test na hodnotu `null` a její blok). Existence obou těchto odpovědností znamená, že metoda dělá více než jen jednu věc. Takže v malé míře porušujeme *princip jediné odpovědnosti*.

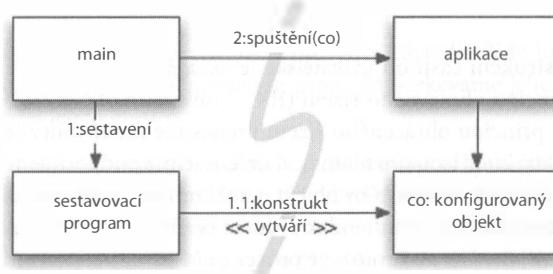
Možná, že tím nejhorším je, že nevíme, zda objekt `MyServiceImpl` je tím správným objektem pro všechny případy. To jsem již naznačoval ve svém komentáři. Proč by měla třída s touto metodou znát globální kontext? Můžeme vůbec někdy znát ty správné objekty, které zde máme používat? Je možné, aby byl jeden typ tím správným pro všechny možné kontexty?

Jeden výskyt odložené inicializace není samozřejmě příliš velkým problémem. V aplikacích však běžně najdeme mnoho podobných instancí malých idiomů určených pro počáteční nastavení. Proto platí, že globální organizační *strategie* (pokud nějaká existuje) je *roztroušena* napříč celou aplikací s malou modularitou a často se značným zdvojením kódu.

Pokud *máme zájem* vytvářet dobré postavené a robustní systémy, neměli bychom nikdy nechat malé, *pohodlné* idiomy narušovat jejich modularitu (těch systémů). Počáteční proces vytváření objektů a jejich propojování není výjimkou. Tento proces bychom měli modularizovat odděleně od logiky běžného provozu a měli bychom se postarat o to, abychom měli globální a konzistentní strategii pro vyřešení hlavních problémů se závislostmi.

## Separování modulu Main

Jedním ze způsobů, jak oddělit vytváření objektů od jejich používání, je jednoduše přesunout všechny záležitosti konstrukce do metody `main` nebo do modulů, které se odtud volají, a zbytek systému navrhout s předpokladem, že všechny objekty byly náležitě vytvořeny a propojeny. (Viz obrázek 11.1.)



Obrázek 11.1. Oddělení konstrukce v modulu `main()`

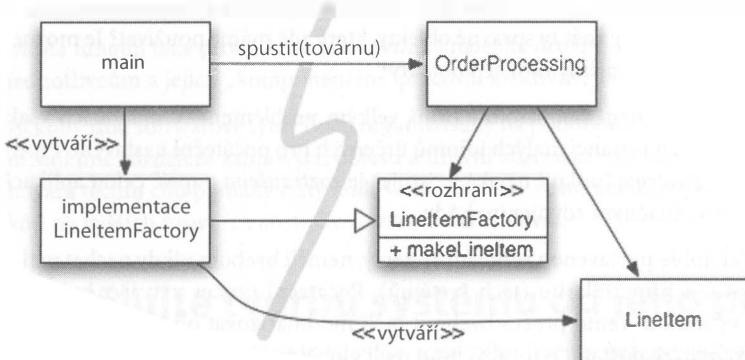
Sledovat tok řízení je jednoduché. Funkce `main` vytvoří objekty, které jsou pro systém nezbytné, a předá je aplikaci, jež je pak používá. Všimněte si směru šipek, představujících závislosti, jak překračují hranici mezi modulem `main` a aplikací. Všechny závislosti vedou jedním směrem – ven z modulu

1. [Mezzaros07].

`main`. To znamená, že aplikace nemá během procesu konstrukce o modulu `main` žádné informace. Prostě předpokládá, že vše bylo vytvořeno správně.

## Továrny

Samozřejmě, že někdy potřebujeme v aplikaci rozhodnout, *kdy* se bude objekt vytvářet. Například v systému pro zpracování objednávek musí aplikace vytvořit instance objektu `LineItem` a přidat je k objektu `Order`. V tomto případě můžeme použít vzor pro abstraktní továrnu (ABSTRACT FACTORY)<sup>2</sup> a nechat aplikaci určit okamžik, *kdy* se bude objekt `LineItem` vytvářet, s tím, že detaily tohoto procesu budou odděleny od aplikačního kódu. (Viz obrázek 11.2.)



Obrázek 11.2. Separace konstrukce s továrnou

Také zde si všimněte, že všechny závislosti vedou z modulu `main` směrem k aplikaci `OrderProcessing`. To znamená, že aplikace je zbavena vazeb na takové detaily, jako je vytváření objektu `LineItem`. Tato vlastnost je součástí `LineItemFactory` Implementation, které je na straně modulu `main`. A opět aplikace řídí vytváření instancí objektu `LineItem` a dokonce poskytuje konstruktoru své specifické argumenty.

## Vkládání závislostí

Výkonný mechanismus pro oddělování konstrukční části od uživatelské je *vkládání závislostí* (DI – Dependency Injection), což je varianta principu obráceného řízení (IoC – Inversion of Control) aplikovaná na správu závislostí.<sup>3</sup> Na základě principu obráceného řízení přenášíme méně důležité odpovědnosti z nějakého objektu na jiné objekty, které jsou pro tento účel určeny, čímž podporujeme princip jedné odpovědnosti. V souvislosti se správou závislostí by objekt neměl mít odpovědnost za vytváření samotných instancí závislostí. Místo toho by měl tuto odpovědnost předat jinému „autoritativnímu“ mechanismu, a tedy by mu měl předat řízení. Protože je proces počátečního nastavení globální záležitostí, je tímto autoritativním mechanismem obvykle buď hlavní program (např. metoda `main`), nebo specializovaný *kontejner*.

Vyhledávání pomocí rozhraní JNDI je „částečnou“ implementací principu vkládání závislostí, kdy objekt žádá adresárový server, aby poskytl „službu“ – zda název vyhovuje daným kritériím.

2. [GOF].

3. Viz například [Fowler].

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

Volající objekt neurčuje, jaký druh objektu se ve skutečnosti vrátí (samozřejmě jen pokud implementuje náležité rozhraní), ale je to volající objekt, který závislost aktivně řeší.

Skutečný princip vkládání závislostí jde ještě o krok dále. Pro vyřešení závislostí nedělá třída žádné přímé kroky a je zcela pasivní. Místo toho poskytuje přístupové metody nebo argumenty konstruktoru (nebo obojí), které se použijí k vložení závislostí. Během procesu konstrukce vytváří kontejner pro vkládání závislostí instanci požadovaných objektů (obvykle na základě požadavku) a používá poskytované argumenty konstruktoru nebo přístupové metody k vzájemnému propojení závislostí. Konfigurační soubor nebo specializovaný konstrukční modul za běhu programu specifikuje, které závislé objekty se ve skutečnosti použijí.

Architektura Spring Framework poskytuje nejznámější kontejner pro vkládání závislostí v jazyce Java.<sup>4</sup> V konfiguračním souboru v XML definujete, které objekty chcete propojit, a pak si vyžádáte konkrétní objekty na základě jména v kódu v Javě. Za chvíli si ukážeme příklad.

Ale co se všemi výhodami odložené inicializace? Někdy je tento idiom společně s principem vkládání závislostí stále použitelný. Za prvé se většina kontejnerů pro vkládání závislostí vytváří až v okamžiku potřeby. Za druhé mnoho z těchto kontejnerů poskytuje mechanismy, které volají továrny nebo konstruují zprostředkující objekty, jež lze použít pro línou inicializaci nebo podobné *optimalizace*<sup>5</sup>.

## Škálování

Velkoměsta vyrůstají z měst a ta zase z osad. Silnice jsou zpočátku úzké a prakticky neexistují, pak se teprve vydláždí a časem se rozšiřují. Malé budovy a parcely poskytnou prostor větším budovám a některé budovy nakonec nahradí mrakodrapy.

Na začátku chybí služby, jako je dodávka energie, vody, kanalizace nebo Internet (ouha!). Tyto služby se také objeví, jakmile vzroste hustota obyvatel a zastavěnost.

Růst města není bezbolevný. Kolikrát jste jeli po „opravované“ silnici téměř na nárazníku jiného řidiče a ptali se sami sebe: „Proč to nepostavili dostačně široké hned napoprvé?“

Jenže takovou úpravu nelze provádět kdykoliv. Kdo by byl schopen zdůvodnit stavbu šestiproudé dálnice přes malé město jen proto, že očekáváme jeho růst? Kdo by chtěl, aby taková silnice vedla přes jejich město?

Víra, že můžeme mít systémy „hned napoprvé“ je mýtem. Místo toho bychom měli realizovat zadání aktuální k dnešnímu dni, pak kód refaktorovat a rozšířit systém tak, aby bylo zítra možno realizovat nové myšlenky. To je základem iterativní a inkrementální práce. Vývoj řízený testy, refaktorování a takto vzniklý čistý kód se o to postarájí na úrovni kódování.

A co na systémové úrovni? Nevyžaduje architektura systému přípravné plánování? Určitě. Ta přece nemůže růst inkrementálně od jednoduššího ke složitějšímu, že?

*Softwarové systémy jsou ve srovnání s fyzickými systémy jedinečné. Jejich architektura může růst inkrementálně, pokud se nám podaří adekvátně oddělit jednotlivé zájmové oblasti.*

4. Viz [Spring]. Rovněž existuje architektura Spring.NET.

5. Mějte na paměti, že odložená inicializace/vyhodnocení je jen optimalizací a možná že je předčasná!

Jak uvidíme, krátkodobý život softwarových systémů to umožnuje. Podívejme se nejdříve na prototypický příklad architektury, která jednotlivé oblasti neodděluje tak, jak by měla. Původní architektury EJB1 a EJB2 zájmové oblasti dobře neoddělovaly, a tudíž svému organickému růstu kladly zbytečné překážky. Podívejte se v *Entity bean* na třídu Bank. Entita bean reprezentuje relační data uložená v paměti. Jinými slovy se jedná o jeden řádek tabulky.

Za prvé musíte definovat lokální (v procesu) nebo vzdálené (oddelené JVM) rozhraní, které by klient mohl použít. Výpis 11.1 ukazuje jedno z možných lokálních rozhraní:

#### **Výpis 11.1. Lokální rozhraní EJB2 pro EJB komponentu 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;
}
```

Ukázal jsem několik atributů adresy banky a kolekci účtů, které tato třída vlastní. Data každého z nich by se zpracovávala v samostatné komponentě EJB Account. Výpis 11.2 ukazuje odpovídající implementaci třídy pro komponentu Bank.

#### **Výpis 11.2. Implementace odpovídající entitní komponenty EJB2**

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;
public abstract class Bank implements javax.ejb.EntityBean {
    // Obchodní logika...
    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 kontejneru EJB
public abstract void setId(Integer id);
public abstract Integer getId();
public Integer ejbCreate(Integer id) { ... }
public void ejbPostCreate(Integer id) { ... }
// Zbytek musel být implementován, ale byly obyčejně prázdné
public void setEntityContext(EntityContext ctx) {}
public void unsetEntityContext() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbLoad() {}
public void ejbStore() {}
public void ejbRemove() {}
}
```

Neukázal jsem zde odpovídající rozhraní *LocalHome*, což je v podstatě továrna pro vytváření objektů, ani žádnou z dotazovacích metod třídy *Bank*, které byste mohli do systému přidat.

Nakonec jste museli napsat jeden nebo několik instalačních popisovačů v XML, které specifikují objektově-relační detailly vzájemné korespondence s úložištěm dat, požadované transakční chování, bezpečnostní omezení atd.

Obchodní logika je těsně svázána s aplikačním „kontejnerem“ EJB2. Musíte vytvořit podtřídy kontejnerových typů a musíte poskytnout mnoho metod pro správu životního cyklu, které kontejner vyžaduje.

Díky těmto vazbám na důležitý Kontejner je izolované jednotkové testování obtížné. Je nezbytné vytvořit napodobeninu kontejneru, což je těžké, nebo strávit mnoho času na nasazení architektury EJB a testů na skutečném serveru. Díky těsným vazbám je využití mimo architekturu EJB2 prakticky nemožné.

Nakonec jsou zde podkopány zásady objektově orientovaného programování. Jedna komponenta bean nemůže dědit od jiné. Všimněte si logiky pro přidávání nového účtu. V architektuře EJB2 je běžně definovat „objekty pro přenos dat“ (DTO – data transfer object), které jsou v podstatě strukturami a nemají definovanou funkčnost. To vede obvykle k redundantním typům, které obsahují v zásadě stejná data, což vyžaduje pro kopírování dat z jednoho objektu na druhý standardizovaný kód.

## Průnik zájmů

V některých oblastech se architektura EJB2 přibližuje skutečnému oddělení zájmových oblastí. Například požadované transakční zpracování, bezpečnost a některé funkce pro uchování dat jsou deklarovány v instalačních deskriptorech a nezávisle na zdrojovém kódu.

Všimněte si, že *zájmové oblasti*, jako je uchování dat, mají tendenci procházet napříč přirozenými objektovými hranicemi domén. Chcete uchovat všechny své objekty a k tomu použijete na více místech stejnou strategii. Například použitím konkrétního databázového systému místo nestrukturovaných souborů, dodržováním určitých konvencí pro tvorbu jmen tabulek a sloupců, použitím konsistentní transakční sémantiky atd.

Můžete v podstatě argumentovat o své strategii uchování dat ve smyslu modularity a zapouzdření. Ale v praxi musíte umístit týž kód, který tuto strategii implementuje, do mnoha objektů. Pro toto činnost používáme termín *průnik zájmů*. Znovu připomínám, že struktura kódu pro uchovávání dat může být modulární a doménová logika, i izolovaně, může být také modulární. Problémem je nízkourovňové křížení těchto domén.

Způsob, jakým zpracovává architektura EJB uchování dat, bezpečnost a transakce v podstatě „předcházela“ *aspektově orientovanému programování* (AOP)<sup>6</sup>, které je obecným pokusem, jak u průniku zájmů obnovit modularitu.

U aspektově orientovaného programování specifikují modulární konstrukty, zvané *aspekty*, takové body systému, ve kterých by měla být nějakým konsistentním způsobem modifikována funkčnost, jež by podporovala nějaký záměr. Tato specifikace se provádí pomocí stručných deklarativních nebo programových mechanismů.

V případě uchování dat byste deklarovali, které objekty a atributy (nebo vzory) se mají uchovávat, a poté byste vašemu rámci pro uchování dat delegovali danou úlohu. Modifikace funkčnosti se provádějí na cílovém kódu aplikačním rámcem AOP neinvazivně<sup>7</sup>. Podívejme se na tři aspekty nebo aspektové mechanismy v Javě.

## Zprostředkovatel v Javě

Zprostředkovatelé (proxy) v Javě jsou vhodné pro jednoduché situace, jako je volání obalových metod v jednotlivých objektech nebo třídách. Avšak dynamické proxy v JDK pracují jen s rozhraními. Abyste vytvořili proxy třídy, musíte použít knihovnu pro zpracování bajtového kódu, jako je CGLIB, ASM nebo Javassist<sup>8</sup>.

Výpis 11.3 ukazuje skeleton proxy JDK, který poskytuje podporu pro uchování údajů v aplikaci Bank, zahrnující jen přístupové metody pro získávání a nastavování seznamu účtů.

### Výpis 11.3. Příklad proxy v JDK

```
// Bank.java (potlačení názvů balíčků...)
import java.util.*;
// Abstrakce třídy bank.
public interface Bank {
    Collection<Account> getAccounts();
    void setAccounts(Collection<Account> accounts);
```

6. Viz [AOsd], kde najeznete všeobecné informace o aspektech a [AspectJ]] a [Colyer], kde najeznete specifické informace ohledně AspectJ.
7. Což znamená, že je nežádoucí ruční editace cílového kódu.
8. Viz [CGLIB], [ASM] nebo [Javassist].

```
}

// BankImpl.java
import java.util.*;
// "Starý dobrý javový objekt" (POJO) implementující abstrakci.
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" vyžaduje jej proxy API.
public class BankProxyHandler implements InvocationHandler {
    private Bank bank;
    public BankHandler (Bank bank) {
        this.bank = bank;
    }
    // Metoda definovaná v InvocationHandler
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        String methodName = method.getName();
        if (methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
        } else if (methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[0]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
        } else {
            ...
        }
    }
    // Zde je mnoho detailů:
    protected Collection<Account> getAccountsFromDatabase() { ... }
    protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}
// Někde jinde...
Bank bank = (Bank) Proxy.newProxyInstance(
    Bank.class.getClassLoader(),
    new Class[] { Bank.class },
    new BankProxyHandler(new BankImpl()));
```

Definovali jsme rozhraní `Bank`, které bude obaleno pomocí proxy, a také jsme definovali „Starý dobrý javový objekt“ (POJO – Plain-Old Java Object) `BankImpl`, který implementuje obchodní logiku. (Na objekt POJO se podíváme za chvíli.)

Proxy API vyžaduje objekt `InvocationHandler` a volá ho. Tím implementuje jakékoli volání metody směrem k proxy. Objekt `BankProxyHandler` používá javovou reflexi API a mapuje volání generické metody na odpovídající metodu v objektu `BankImpl`, atd.

Je *mnoho* relativně komplikovaného kódu, dokonce i v takto jednoduchém případě<sup>9</sup>. Dalším problémem je používání některé z knihoven umožňujících manipulaci s bajty. Tento „objem“ kódů a jeho komplikovanost jsou dva nedostatky proxy. Komplikují vytváření čistého kódu! Neposkytují také mechanismy pro určení připojních bodů v rámci celého systému, což je pro skutečné řešení AOP zapotřebí<sup>10</sup>.

## Čisté rámce Java AOP

Naštěstí může být většina standardizovaného kódu zprostředkovatelů (proxy) zpracována automaticky pomocí nástrojů. V několika rámcích Javy se zprostředkovatele používají vnitřně. Například Spring AOP a JBoss AOP a implementují aspekty v čisté Javě<sup>11</sup>. V případě rámce Spring píšete vaši obchodní logiku jako *staré dobré javové objekty* (POJO). POJO jsou výlučně zaměřeny na svou doménu. Nemají žádné závislosti na podnikových rámcích (nebo na jakýchkoliv jiných doménách). To znamená, že jsou koncepcně jednodušší a snadněji se testují. Díky jejich relativní jednoduchosti snadněji zabezpečíte implementaci, která odpovídá požadavkům zákazníka, a zachováte nebo vyvíjíte další kód podle budoucích požadavků.

Infrastrukturu požadované aplikace, včetně křížení zájmů, jako je uchování dat, transakce, bezpečnost, použití vyrovnavacích pamětí, zálohování počítačů atd., můžete zabudovat pomocí deklarativních konfiguračních souborů nebo API. V mnoha případech vlastně specifikujete knihovní aspekty Spring nebo JBoss, ve kterých rámec zpracovává pro uživatele mechanismy používání proxy z Javy nebo knihovny bajtového kódu transparentně. Tyto deklarace usměrňují činnost kontejneru pro vkládání závislostí (DI – dependency injection), které vytváří instance hlavních objektů a podle požadavků je vzájemně propojuje.

Výpis 11.4 ukazuje typický fragment konfiguračního souboru rámce Spring V2.5 - `app.xml`<sup>12</sup>.

### Výpis 11.4. Konfigurační soubor Spring 2.X

```
<beans>
    ...
    <bean id="appDataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
```

9. Podrobnější příklady proxy API a příklady jeho využití, viz například [Goetz].

10. Někdy si AOP pleteme s technikami, které ho implementují, jako je třeba metoda *zachycení* a „obalení“ pomocí proxy. Skutečná hodnota systému AOP je v jeho schopnosti specifikovat systémovou funkčnost stručným a modulárním způsobem.

11. Viz [Spring] a [JBoss]. „Čistá Java“ znamená, že se používá bez AspectJ.

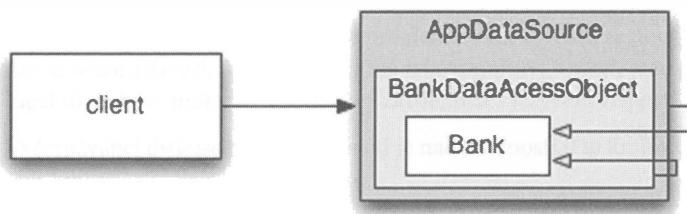
12. Převzato a upraveno z <http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25>.

```

<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:accessObject-ref="bankDataAccessObject"/>
...
</beans>

```

Každá komponenta („bean“) je jako jedna část matrjošky s doménovým objektem pro třídu Bank, obalenou objektem pro přístup k datům (DAO – Data Access Object), který je sám obalen zdvojem dat ovladače JDBC. (Viz obrázek 11.3.)



Obrázek 11.3. Matrjoška dekorátorů

Klient věří, že volá metodu `getAccounts()` objektu `Bank`, ale ve skutečnosti komunikuje s úplně vnějším objektem z množiny DEKORÁTORŮ<sup>13</sup>, který rozšiřuje základní funkčnost objektu `Bank` (POJO). Mohli bychom přidat další dekorátory pro transakce, vyrovnávací paměti a tak dále. V aplikacích je zapotřebí jen několik rádků na to, aby byl vznesen dotaz kontejneru pro vkládání závislostí na objekty nejvyšší úrovně systému, jak to specifikuje soubor v XML.

```

XmlBeanFactory bf =
new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");

```

Protože je zapotřebí jen velmi málo řádků kódu v Javě specifického pro rámcem Spring, je *aplikace zbavená téměř všech vazeb na rámcem Spring*. Taktto jsou eliminovány problémy s těsným provázáním, které vidíme v EJB2.

Ačkoliv jazyk XML může být upovídáný a obtížně čitelný<sup>14</sup>, „zásada“, specifikovaná v těchto konfiguračních souborech, je jednodušší než komplikovaná logika proxy a aspektů, která je skryta před našimi zraky a jež se vytváří automaticky. Tento typ architektury má natolik donucovací účinek, že rámcem, jako je Spring, vedly ke kompletnímu přepracování standardů EJB ve verzi 3. EJB3 se do znač-

13. [GOF].

14. Příklad lze zjednodušit za použití mechanismu, která využívá pravidlo konvence má přednost před konfigurací a anotace z Javy 5, aby se zredukoval objem nutné explicitní „spojovací“ logiky.

né míry drží modelu rámce Spring pokud jde o deklarativní podporu průniků zájmů a používání konfiguračních souborů XML a/nebo anotací z Javy 5.

Výpis 11.5 ukazuje objekt Bank po přepsání do EJB3<sup>15</sup>.

#### **Výpis 11.5. Objekt Bank v 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 // Objekt na jednom řádku s dabázovým řádkem objektu 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) {
        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;
    }
}
```

Tento kód je mnohem čistější, než jeho originál v EJB2. Některé detaily jeho entit jsou obsaženy v anotacích. Protože žádná z těchto informací není mimo anotace, je kód čistý, jasný, a tudíž ho lze snadno testovat, udržovat atd.

---

15. Upraveno podle <http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html>.

Některé nebo veškeré informace ohledně uchování dat v anotacích lze přesunout do instalačních deskriptorů napsaných v XML, pokud je to žádoucí, čímž získáme opravdu čisté objekty POJO. Pokud se mapování detailů, týkajících se uchování dat, nebude příliš často měnit, mnoho týmů může dát přednost alternativě anotace zachovat, ale ve srovnání s invazivitou EJB2 s mnohem menším počtem nedostatků.

## Aspekty AspectJ

Nejúplnejší nástroj pro separaci zájmových oblastí pomocí aspektů je jazyk AspectJ<sup>16</sup>, což je rozšíření Javy, který poskytuje „prvotřídní“ podporu pro aspekty jako konstrukty modularity. Přístup založený na čistém jazyce Java, poskytovaný rámci Spring AOP a JBoss AOP, vyhoví v 80–90 procentech případů, ve kterých jsou aspekty nejužitečnější. Avšak AspectJ poskytuje velmi bohatou množinu nástrojů pro separaci zájmových oblastí. Nevýhodou jazyka AspectJ je, že si musíte osvojit několik nových nástrojů, naučit se nové jazykové konstrukty a uživatelské idiomu.

Problémy osvojování nových nástrojů byly částečně zmírněny nedávným uvedením „anotačního formuláře“ do nástroje AspectJ, kde se pro definování aspektů za použití čistého javového kódu používají anotace z Javy 5. Rámec Spring má také množství funkcí, které velmi zjednoduší začleňování aspektů na bázi anotace i týmům bez zkušenosti s jazykem AspectJ.

Vyčerpávající diskuse na téma AspectJ je nad možnosti této knihy. Chcete-li vědět více, podívejte se na [AspectJ], [Colyer] a [Spring].

## Testování systémové architektury

Vliv separace zájmových oblastí pomocí aspektů nelze přečeňovat. Pokud můžete napsat logiku své aplikační domény za pomoci objektů POJO, které jsou na úrovni kódu, bez jakýchkoliv vazeb na zájmové oblasti architektury, pak je možné opravdu vaši architekturu *vyzkoušet*. Můžete postupovat směrem od jednoduchého k obtížnějšímu a podle potřeb a poptávky používat nové technologie.

Není nezbytně nutné používat „vodopádový model“ (BDUF – Big Design Up Front)<sup>17</sup>. V praxi je vodopádový model dokonce škodlivý kvůli tomu, že díky psychologickému odporu k úsilí něco rušit znemožňuje reagovat na změny a také kvůli způsobu, jakým volba architektury ovlivňuje následné promýšlení návrhu.

Architekti budov vodopádový model používat musejí, protože není možné dělat radikální architektonické změny na velké a reálné stavbě, pokud jsou stavební práce již v plném proudu<sup>18</sup>. Ačkoliv má software svou vlastní *fyziku*<sup>19</sup>, jsou radikální změny ekonomicky možné, *pokud* struktura softwaru dokáže efektivně oddělit jednotlivé zájmové oblasti.

16. Viz [AspectJ] a [Colyer].

17. Abychom tento postup nezaměňovali s dobrou praxí vytvořit návrh dopředu, BDUF znamená navrhovat vše dopředu před jakoukoliv implementací čehokoliv.

18. Stále ještě existuje významný podíl opakovaného zkoumání a diskuse nad detaily, i když stavba již začala.

19. Termín *softwarová fyzika* byl poprvé použit v [Kolence].

To znamená, že softwarový projekt můžeme začít s „naivní a jednoduchou“ architekturou, která ale musí být zbavena vazeb a musí koncovému uživateli rychle poskytovat výsledky jeho zadání. S dalším rozširováním systému pak můžeme přidávat další infrastrukturu. Některé z nejvýznamnějších světových webových serverů dosahly vysoké úrovně dostupnosti a výkonu za použití rafinovaného použití vyrovnávacích pamětí, bezpečnosti, virtualizace atd. To vše bylo provedeno efektivně a pružně, protože jejich projekty byly provedeny s minimem propojení a na každé úrovni abstrakce a rozsahu byly přiměřeně **jednoduché**.

To samozřejmě neznamená, že se pustíme do projektu „bezhlavě“. Máme nějaká obecná očekávání, cíle a časový rozvrh projektu, stejně jako i obecnou strukturu výsledného systému. Musíme si však stále zachovat možnost měnit postup podle toho, jak se mění okolnosti.

Prvotní architektura EJB je jen jedním z mnoha dobré známých API, které jsou připraveny s přemírou pečlivosti, jež škodí, a které jsou kompromisem v otázce separace zájmových oblastí. Dokonce i dobře navržené rozhraní API může být čímsi nad míru v případě, že není nutně zapotřebí. Dobré rozhraní API by mělo po většinu času zmizet ze zorného pole, aby mohl tým zaměřit většinu svého tvůrčího úsilí na implementaci uživatelského zadání. Pokud tomu tak nebude, budou architektonická omezení znesnadňovat efektivní plnění závazků vůči zákazníkovi.

Abych rekapituloval toto dlouhé pojednání:

*Optimální architektura systému se skládá z modularizovaných zájmových domén, které jsou všechny implementovány pomocí objektů POJO (nebo nějakých jiných). Různé domény jsou integrované s Aspekty nebo podobnými nástroji, které jsou co nejméně invazivní. Tato architektura může být řízena testy stejně jako její kód.*

## Optimalizujte rozhodování

Modularita a separace zájmových oblastí umožňují decentralizované řízení a rozhodování. V do- statečně velkém systému, ať už se jedná o město nebo o softwarový projekt, nikdo nemůže rozhodovat sám.

Všichni dobře víme, že nejlepší je dát odpovědnost těm nejkvalifikovanějším osobám. Často zapomínáme na to, že je také nejlepší *odkládat rozhodnutí až na poslední možný okamžik*. To není lenost nebo nezodpovědnost. To nám umožňuje provádět kvalifikované volby s nejlepšími možnými informacemi. Předčasná rozhodnutí jsou rozhodnutí s méně kvalifikovanou znalostí. Budeme-li se rozhodovat příliš brzy, budeme mít slabší zpětnou vazbu od zákazníka, horší vnitřní reflexi projektu a méně zkušeností s implementacemi různých variant.

*Flexibilita poskytovaná systémem POJO s modularizovanými zájmovými oblastmi nám umožňuje přijímat optimální a včasná rozhodnutí, založená na nejposlednějších znalostech. Složitost těchto rozhodnutí je rovněž nižší.*

## Používejte rozumně standardy, pokud přinášejí prokazatelnou hodnotu

Je nádherné sledovat stavbu nových budov pro rychlosť, s jakou se staví (dokonce i v tuhé zimě) a kvůli pozoruhodným konstrukcím, které umožňuje dnešní technologie. Stavebnictví je vyspělý průmysl s vysoko optimalizovanými částmi, metodami a standardy, které se vyvinuly pod tlakem staletí.

Mnoho týmů používalo architekturu EJB2, protože to byl standard, i když by stačilo použít jednodušší a méně komplikované návrhy. Viděl jsem týmy, které byly uneseny různými *bombasticky vychvalovanými* standardy, jak ztrácejí ze zřetele realizaci hodnot pro své zákazníky.

*Standardy usnadňují opakování používání myšlenek a komponent, získávají lidi s odpovídajícími zkušenostmi, zapouzdřují dobré myšlenky a spojují dohromady komponenty. Avšak proces vytváření standardů někdy může trvat příliš dlouho, zatímco průmysl čeká a některé standardy ztrácejí kontakt se skutečnými potřebami těch, již je přebírají a kterým mají sloužit.*

## Systémy potřebují doménově specifické jazyky

Budování staveb, jako většina domén, stála u vývoje bohatého jazyka se slovní zásobou, idiomu a vzory<sup>20</sup>, které dokážou jasně a stručně tlumočit základní informace. V oblasti softwaru se nedávno obnovil zájem na vytvoření *doménově specifických jazyků* (DSL – Domain-Specific Languages)<sup>21</sup>, což jsou samostatné, malé skriptovací jazyky nebo API ve standardních jazycích, které umožňují, aby byl kód napsán takovým způsobem, že je lze číst jako strukturovaný druh prózy, jak by ho mohl napsat znalec domény.

Dobrý doménově specifický jazyk minimalizuje „komunikační mezeru“ mezi doménovým konceptem a kódem, který jej implementuje tak, jako aktivní cvičení optimalizuje komunikaci mezi členy týmu a dalšími účastníky projektu. Pokud implementujete doménovou logiku ve stejném jazyce, který používá doménový odborník, je zde menší riziko, že bude doménu realizovat chybně.

Jsou-li doménově specifické jazyky používány efektivně, vyzdvihou úroveň abstrakce nad idiomu kódu a návrhové vzory. Vývojářům umožní na odpovídající úrovni abstrakce odhalit záměr kódu.

*Doménově specifické jazyky umožňují, aby všechny úrovně abstrakce a všechny domény v aplikaci byly vyjádřeny formou POJO (Plain-Old Java Object) počínaje politikou vysoké úrovne a konče detailem nízké úrovne.*

## Závěr

I systémy musí být čisté. Invazivní architektura zdolává doménovou logiku a má dopad na flexibilitu. Pokud je doménová logika obskurní, trpí tím kvalita, protože chyby budou lépe skryty a zadání uživatele se bude realizovat hůře. Pokud uděláme kompromis v otázce flexibility, bude to na úkor produktivity a přínos přijde vniveč.

Na všech úrovních abstrakce by měl být jasný záměr. Toho dosáhnete jen tehdy, budete-li používat objekty POJO a aspektové mechanismy, abyste neinvazivně zahrnuli jiné implementace zájmových oblastí.

Ať už navrhujete systémy nebo individuální moduly, mějte vždy na paměti, že máte používat to nej-jednodušší, co by mohlo fungovat.

20. Práce Christophera Alexandra [Alexander] měla na softwarovou komunitu zvlášt velký vliv.

21. Viz například [DSL]. [JMock] je dobrým příkladem javového rozhraní API, které vytváří DSL.

## Použitá literatura

- [Alexander]: Christopher Alexander, *A Timeless Way of Building*, Oxford University Press, New York, 1979.
- [AOSD]: Aspect-Oriented Software Development port, <http://aosd.net>
- [ASM]: ASM Home Page, <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, NJ, 2005.
- [DSL]: Domain-specific programming language, [http://en.wikipedia.org/wiki/Domain-specific\\_programming\\_language](http://en.wikipedia.org/wiki/Domain-specific_programming_language)
- [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]: Javassist Home Page, <http://www.csg.is.titech.ac.jp/~chiba/javassist/>
- [JBoss]: JBoss Home Page, <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, pp. 1024–1040, 1972.
- [Spring]: *The Spring Framework*, <http://www.springframework.org>
- [Mezzaros07]: *XUnit Patterns*, Gerard Mezzaros, Addison-Wesley, 2007.
- [GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

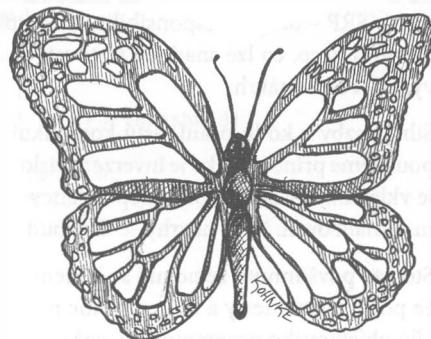
# KAPITOLA 12

## Vývoj

*Jeff Langr*

### **V této kapitole najdete:**

- ◆ Čistota pomocí vyvíjejícího se návrhu
- ◆ První pravidlo jednoduchého návrhu: Projde všemi testy
- ◆ Zásady jednoduchého návrhu 2–4: refaktorování
- ◆ Žádný zdvojený kód
- ◆ Expresivita
- ◆ Minimální třídy a metody
- ◆ Závěr
- ◆ Použitá literatura



## Čistota pomocí vyvíjejícího se návrhu

Co kdyby existovala čtyři jednoduchá pravidla, kterými byste se mohli řídit a jež by vám pomáhala s vytvářením dobrého návrhu? Co kdybyste dodržováním těchto pravidel získali náhled do struktury a návrhu vašeho kódu, což by usnadnilo použití principů, jako je princip jediné odpovědnosti nebo princip inverze závislosti? Co kdyby tato čtyři pravidla *vznik* dobrých návrhů usnadnila?

Mnoho z nás má pocit, že čtyři pravidla Kenta Becka *Jednoduchého návrhu*<sup>1</sup> mohou při vytváření dobré navrženého softwaru významně pomáhat.

Podle Kenta je návrh „jednoduchý“ tehdy, když vyhovuje těmto zásadám:

- ◆ Projde všemi testy.
- ◆ Neobsahuje žádné zdvojení.
- ◆ Vyjadřuje záměr programátora.
- ◆ Minimalizuje počet tříd a metod.

Tato pravidla jsou uvedena v pořadí své důležitosti.

### První pravidlo jednoduchého návrhu: Projde všemi testy

Na prvním místě musí návrh vyústit v systém, který funguje tak, jak má. Systém může na papíře vypadat perfektně, ale neexistuje žádný jednoduchý způsob, jak ověřit, že systém doopravdy funguje tak, jak má. Pak je jakékoli papírové úsilí sporné.

Systém, který je důkladně otestován a projde kdykoliv testy, je testovatelný systém. To je samozřejmost, ale je velmi důležitá. Systémy, které nejsou testovatelné, nejsou verifikovatelné. Systém, který nelze verifikovat, by neměl být nikdy nasazen.

Naštěstí úsilí, které směřuje k testovatelným systémům, nás zároveň vede k návrhu s malými a jednoúčelovými třídami. Je prostě jednoduší testovat třídy, které vyhovují principu jedné odpovědnosti (SRP – Single Responsibility Principle). Čím více napišeme testů, tím více se budeme snažit vytvářet něco, co lze snadněji testovat. Takže úsilí vytvořit plně testovatelný systém nám pomáhá vytvářet lepší návrh.

Silné vazby v kódu psaní testů komplikuje. Podobně platí, že čím více testů napišeme, tím častěji použijeme principy, jako je inverze závislosti (DIP – Dependency Inversion Principle) a nástroje, jako je vkládání závislostí (DI – Dependency Injection), rozhraní a abstrakce, abychom vzájemné vazby minimalizovali. Naše návrhy se dále budou zlepšovat.

Stojí za povšimnutí, že pokud se budeme řídit jednoduchým a samozřejmým pravidlem, které říká, že potřebujeme testy a že je musíme neustále spouštět, bude to mít vliv na dodržování primárních cílů objektového programování, což znamená slabé vazby a vysokou soudržnost.

1. [XPE].

## Zásady jednoduchého návrhu 2–4: refaktorování

Máme-li konečně testy, máme možnost udržovat kód a třídy čisté. Toho dosáhneme postupným refaktorováním kódu. Po každém novém řádku, který do kódu přidáme, se zastavíme a ohlédneme se zpět na nový návrh. Nezhoršili jsme jej? Pokud ano, vycistíme jej a spustíme testy, abychom si ukázali, že jsme nic nepoškodili. *Skutečnost, že máme k dispozici testy, eliminuje obavy, že během čistění kódu něco zkazíme!*

Během tohoto refaktorování můžeme použít cokoliv z velkého množství znalostí pro tvorbu dobrého návrhu softwaru. Můžeme zvyšovat soudržnost, snižovat stupeň vazeb, oddělovat zájmové oblasti, modularizovat systémové části kódu, zmenšovat funkce a třídy, volit lepší názvy atd. Zde také aplikujeme poslední tři pravidla jednoduchého návrhu: eliminovat zdvojený kód, zabezpečit expresivitu a minimalizovat počet tříd a metod.

## Žádný zdvojený kód

V dobře navrženém systému je zdvojený kód úhlavním nepřítelem. Představuje práci navíc, riziko navíc a další zbytečnou složitost. Zdvojený kód se projevuje v mnoha formách. Řádky kódu, které vypadají naprosto shodně, jsou samozřejmě zdvojené. Řádky kódu, které si jsou podobné, mohou být přepracovány ještě do větší podobnosti a lze je lépe refaktorovat. A zdvojování může mít mnoho dalších forem, jako opakovaná implementace. Mohli bychom mít například dvě metody ve třídě kolekce:

```
boolean isEmpty() {}
```

Mohli bychom mít pro každou metodu oddělené implementace. Metoda `isEmpty` by mohla zaznamenávat logickou hodnotu, zatímco metoda `size` by mohla zaznamenávat hodnotu čítače. Nebo bychom toto zdvojení mohli odstranit tím, že svážeme metodu `isEmpty` s definicí `size`:

```
boolean isEmpty() {
    return 0 == size();
}
```

Vytváření čistých systémů vyžaduje vůli po eliminaci zdvojeného kódu, byť by to mělo být jen na několika řádcích. Podívejte se například na následující kód:

```
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;
}
Abychom byl tento systém čistý, měli bychom eliminovat malé zdvojení kódu
v metodách scaleToOneDimension 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;
}

```

Jakmile extrahujeme tento nepatrny objem shodného kódu, všimneme si porušení principu jedné odpovědnosti (SRP). Mohli bychom tedy přesunout extrahovanou novou metodu do jiné třídy. To zvýší její viditelnost. Někdo jiný z týmu by mohl zpozorovat další možnosti abstrakce nové metody a používat ji v novém kontextu. Tohle „znovupoužití v malém“ může způsobit, že se složitost systému značně sníží. Pochopení způsobu, jak dosáhnout dalšího použití v malém měřítku, je základem pro dosažení téhož ve velkém.

Vzor šablonová metoda (TEMPLATE METHOD)<sup>2</sup> je běžnou technikou, jak odstranit zdvojený kód na vyšší úrovni. Například:

```

public class VacationPolicy {
    public void accrueUSDivisionVacation() {
        // kód pro výpočet dovolené na základě odpracovaných hodin k danému datu
        // ...
        // kód, který má zajistit, že dovolená vyhoví minimu platnému v USA
        // ...
        // kód pro žádost o dovolenou na záznam výplatní listiny
        // ...
    }
    public void accrueEUDivisionVacation() {
        // kód pro výpočet dovolené na základě odpracovaných hodin k danému datu
        // ...
        // kód, který má zajistit, že dovolená vyhoví minimu platnému v USA
        // ...
        // kód pro žádost o dovolenou na záznam výplatní listiny
        // ...
    }
}

```

2. [GOF].

Kód, obsažený v metodách `accrueUSDivisionVacation` a `accrueEuropeanDivisionVacation` je do značné míry shodný, až na výjimku, ve které se vypočítávají zákonná minima. Tato část algoritmu se mění podle typu zaměstnance.

Zřejmě zdvojení můžeme odstranit použitím šablonové metody.

```
abstract public class VacationPolicy {  
    public void accrueVacation() {  
        calculateBaseVacationHours();  
        Expressive 175  
        alterForLegalMinimums();  
        applyToPayroll();  
    }  
    private void calculateBaseVacationHours() { /* ... */ };  
    abstract protected void alterForLegalMinimums();  
    private void applyToPayroll() { /* ... */ };  
}  
public class USVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // US specific logic  
    }  
}  
public class EUVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
        // EU specific logic  
    }  
}
```

Podtřídy vyplní „mezeru“ v algoritmu `accrueVacation` a poskytnou jediné kousky informace, které nejsou zdvojené.

## Expresivita

Většina z nás již má zkušenosť a pracovala s komplikovaným kódem. Mnozí z nás nějaký ten spletitý kód sami vytvořili. Je jednoduché psát kód, kterému *my* rozumíme, protože v době jeho psaní jsme do problému zasvěceni a rozumíme všemu, co se pokoušíme vyřešit. Ostatní, kteří budou s kódem pracovat, nebudou nikdy problému rozumět tak dobře.

Největší část nákladů na softwarové projekty jde na vrub dlouhých lhůt na jeho údržbu. Abychom minimalizovali možnost závad, které mohou vzniknout spolu se zaváděním změn, je pro nás zásadní věcí vědět, co systém dělá. Se vztuštající složitostí systémů zabírá vývojářům víc a víc času pronikání do jeho zákoutí a zvyšuje se možnost nedorozumění. Kód by tedy měl jasně vyjadřovat záměr autora. Čím čistěji umí autor kód vytvořit, tím méně času budou muset trávit ostatní na tom, aby jej pochopili. To bude snižovat počet chyb a sniží se i náklady na údržbu.

Můžete se vyjadřovat pomocí správné volby názvů. Když uslyšíme název třídy nebo funkce, neměli bychom být překvapeni tím, co dělají.

Můžete se rovněž vyjádřit tím, že vaše funkce a třídy budou malé. Malé třídy a funkce se obvykle pojmenovávají lépe, jednodušeji se píšou a je snadnější je pochopit. Můžete se také vyjádřit tím, že použijete standardní terminologii. Například návrhové vzory jsou do značné míry záležitostí komuni-

kace a expresivnosti. Používáním standardních názvů vzorů, jako je například Příkaz (COMMAND) nebo návštěvník (VISITOR) v názvech tříd, které tyto vzory implementují, můžete stručně popsat váš návrh tak, aby byl pochopitelný i ostatním vývojářům.

Dobře napsané testy také dokážou něco sdělit. Primárním cílem testů je být dokumentací, jejíž součástí jsou příklady. Když bude někdo naše testy číst, měl by být schopen rychle pochopit, k čemu daná třída slouží.

Ale nejdůležitějším způsobem, jak něco vyjádřit, je *zkusit to*. Příliš často napíšeme funkční kód a pak přejdeme k řešení dalšího problému, aniž bychom se dostatečně věnovali dostatečně dalšímu úkolu, aby nás kód mohli číst i jiní a neměli s tím těžkosti. Mějte na paměti, že další osobou, která se bude čtením kódu zabývat, budete nejspíše vy sami.

Proto dejte trochu průchod hrドosti na vaši profesionalitu. Věnujte trochu času každé své funkci a třídě. Volte lepší názvy, rozdělujte velké funkce na menší a obecně věnujte pozornost tomu, co jste právě vytvořili. Pěče a starostlivost jsou cenný zdroj.

## Minimální třídy a metody

I pojmy tak fundamentální, jako je eliminace zdvojeného kódu, jeho schopnost se vyjádřit a princip jedné odpovědnosti (SRP – Single Responsibility Principle) by se neměly přehánět. V rámci snahy mít malé třídy i metody jich můžeme vytvořit příliš mnoho. Takže tohle pravidlo naznačuje, že bychom měli zachovat i nízký počet funkcí a tříd.

Někdy může velký počet tříd a funkcí vést k nesmyslnému dogmatismu. Uvažujte například o standardu kódování, který klade důraz na vytváření rozhraní pro všechny třídy. Nebo o vývojářích, kteří trvají na tom, aby každé pole a funkčnost byly separovány do datových a funkčních tříd. Takovým dogmatům byste se měli bránit a prosazovat pragmatičtější přístup.

Naším cílem je mít celkový systém malý i s malými funkcemi a třídami. Mějte ale na paměti, že z výše uvedených pravidel pro jednoduchý návrh má tohle pravidlo tu nejnižší prioritu. Takže ačkoliv je důležité mít nízký počet tříd a funkcí, je důležitější mít testy, eliminovat zdvojení kódu a umět se dobrě vyjadřovat.

## Závěr

Existuje nějaká sada jednoduchých postupů, které by mohly nahradit zkušenosť? Určitě ne. Na druhé straně postupy popsané v této kapitole a v této knize jsou v krystalické podobě desítky let sbíraných zkušenosť, které si autoři ověřili v praxi. Budou-li vývojáři postupy jednoduchého návrhu dodržovat, mohou a budou mít odvahu i schopnosti k tomu, aby si osvojili dobré principy a vzory, které by se jinak museli dlouhá léta učit.

## Použitá literatura

[XPE]: *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison-Wesley, 1999.

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

# KAPITOLA 13

## Souběžnost

*Brett L.Schuchert*

### V této kapitole najdete:

- ◆ Proč souběžnost?
- ◆ Problémy
- ◆ Principy ochrany souběžnosti
- ◆ Vyznejte se ve své knihovně
- ◆ Poznejte své běhové modely
- ◆ Pozor na závislosti mezi synchronizovanými metodami
- ◆ Mějte synchronizované sekce malé
- ◆ Je obtížné napsat korektní kód pro vypínání
- ◆ Testování kódu podprocesů
- ◆ Závěr
- ◆ Použitá literatura



„Objekty jsou abstrakcemi procesů. Podprocesy jsou abstrakcemi plánování.“

James O. Coplien<sup>1</sup>

Psaní čistých programů s paralelními (souběžnými) podprocesy je obtížné – velmi obtížné. Mnohem jednodušší je psát kód, který běží v jediném podprocesu. I psát kód s více podprocesy, který vypadá hezky na povrchu, ale uvnitř obsahuje chyby, je jednodušší. Takový kód funguje dobře až do okamžiku, kdy je systém vystaven nějaké záteži.

V této kapitole budeme probírat potřebu paralelního programování a potíže, které přináší. Ukážeme si několik doporučení, jak se s těmito potížemi vypořádat, a jak napsat čistý souběžný kód. Na závěr se budeme zabývat tematikou, která se týká testování paralelního kódu.

Čistý paralelní kód je složité téma, vhodné pro samostatnou knihu. Naše strategie v této knize je předložit přehled a poskytnout detailnejší návod v dodatku A, v kapitole „Souběžnost II“ na straně 321. Pokud vás zajímá jen souběžnost, pak vám tato kapitola bude stačit. Pokud chcete souběžnosti rozumět podrobněji, měli byste si přečíst také kapitolu „Souběžnost II“.

## Proč souběžnost?

Souběžnost je strategie vedoucí k odstraňování vazeb. Pomáhá nám zbavit se vazeb mezi tím, co se zpracovává, od okamžiku, kdy se to zpracovává. V aplikacích s jedním podprocesem je *co* a *kdy* tak silně svázáno, že stav celé aplikace lze často zjistit prohlídkou a trasováním lokálních proměnných v zásobníku. Programátor, který takový systém ladí, může nastavit bod přerušení nebo jejich sekvenci a znát stav systému podle toho, ve kterém bodě se program zastavil.

Zrušení vazby mezi *co* a *kdy* může zásadně zlepšit jak výkon, tak i strukturu aplikace. Z hlediska struktury vypadá aplikace spíše jako mnoho malých spolupracujících počítačů než jako jeden velký hlavní cyklus. To může zjednodušit chápání systému a nabídnout některé efektivní postupy, jak separovat zájmové oblasti.

Podívejte se například na standardní „servletový“ model webových aplikací. Tyto systémy běží pod deštníkem Webu nebo kontejneru EJB, který částečně řídí souběžnost za vás. Servlety se provádějí asynchronně na základě požadavku, který může přijít z webu kdykoliv. Programátor servletu nemusí ošetřovat všechny příchozí požadavky. V principu žije každý běh servletu ve svém vlastním malém světě a je oddělen od běhu ostatních servletů.

Samozřejmě že kdyby to bylo tak jednoduché, by byla tato kapitola zbytečná. Ve skutečnosti má oddělování, které provádí webový kontejner, k dokonalosti hodně daleko. Programátoři servletů se musí mít velmi na pozoru a musí být velmi opatrní, aby zajistili, že jejich souběžné programy jsou bez chyb. Přesto jsou výhody struktury modelu servletu velmi důležité.

Ale struktura není jediným důvodem pro zavádění souběžnosti. Některé systémy mají stanovenou dobu odezvy a vyžadují výkon, jenž je zapotřebí pro řešení souběžnosti na bázi ručně psaného kódu. Podívejte se například na jeden podproces s informačním agregátorem, který z mnoha různých webových stránek shromažďuje informace a slučuje je do denního přehledu. Protože systém má jen jeden podproces, kontaktuje vždy jen jednu webovou stránku, a než může zahájit další cyklus, musí

1. Soukromá korespondence.

dokončit ten stávající. Jeden denní běh se má provést za méně než 24 hodin. Avšak se vznikem dalších a dalších webových stránek se čas prodlužuje, až hodnotu 24 hodin přesáhne. Během své činnost stráví podproces značný čas čekáním na webové sokety, aby mohlo dokončit vstupní/výstupní operace. Pokud použijeme algoritmus s více podprocesy, můžeme se připojit na více webových stránek najednou a můžeme výkon zlepšit.

Nebo uvažujte o systému, který zpracovává požadavky jen od jednoho uživatele a zpracování každého z nich trvá jen jednu sekundu. Tento systém je schopen odpovídat několika uživatelům, ale s rostoucím počtem uživatelů roste i doba odezvy systému. Žádný uživatel se nebude chtít rádit do fronty za dalších sto padesát jiných. Ale při zpracování více uživatelů najednou bychom mohli dobu odezvy systému zkrátit.

Nebo se podívejte na systém, který analyzuje velké objemy dat, ale může poskytnout celkové řešení až po jejich zpracování. Možná, že by bylo možné každou množinu dat zpracovat na jiném počítači, velké objemy dat by se zpracovávaly paralelně.

## Mýty a mylné názory

Máme tedy přesvědčivé důvody pro zavedení souběžných procesů. Jak jsme si však řekli dříve, souběžnost programů je obtížná záležitost. Nebudete-li opravdu opatrni, můžete způsobit velmi nepříjemné situace. Podívejme se na obvyklé mýty a mylné názory:

- ◆ *Souběžnost vždy zlepší výkon*

Někdy souběžnost může zvýšit výkon, ale pouze tehdy, když existují dlouhé čekací doby, které lze sdílet mezi jednotlivými podprocesy nebo paralelními procesory. Ani jedno z toho není trixiální.

- ◆ *Během psaní souběžných programů se návrh nemění*

Ve skutečnosti může být návrh souběžných programů značně odlišný od návrhu systému s jedním procesem. Oddělení otázky *co* od otázky *kdy* má obvykle na strukturu systému zásadní vliv.

- ◆ *Pracujete-li s kontejnery, jako jsou Web nebo EJB, nemusíte rozumět problémům souběžnosti.*

V praxi bude lepší, když budete znát činnost svého kontejneru, jak jej zajistit před nebezpečím souběžné aktualizace a před zablokováním, které je popsáno dále v této kapitole.

Zde je několik dalších rozumných rad, které se týkají psaní souběžného softwaru:

- ◆ *Souběžnost znamená nějakou práci navíc, jak co do výkonu, tak i do psaní dalšího kódu.*

- ◆ *Správná souběžnost je složitá i v případě jednoduchých problémů.*

- ◆ *Chyby vzniklé při souběžném chodu programů nejsou obvykle opakovatelné, takže jsou často ignorovány jako jednorázové události<sup>2</sup> a ne jako chyby, kterými skutečně jsou.*

- ◆ *Souběžnost vyžaduje často zásadní změny strategie návrhu.*

---

2. Kosmické paprsky, závady atd.

## Problémy

Proč je paralelní programování tak obtížné? Podívejte se na následující triviální třídu:

```
public class X {
    private int lastIdUsed;
    public int getNextId() {
        return ++lastIdUsed;
    }
}
```

Řekněme, že vytvoříme instanci třídy `X`, nastavíme pole `lastIdUsed` na hodnotu 42 a pak budeme tuto instanci sdílet mezi dvěma podprocesy. Předpokládejme nyní, že oba tyto podprocesy volají metodu `getNextId()`. Existují tři možné výstupy:

- ◆ První podproces bude mít hodnotu 43, druhý 44 a hodnota proměnné `lastIdUsed` bude 44.
- ◆ První podproces bude mít hodnotu 44, druhý 43 a hodnota proměnné `lastIdUsed` bude 44.
- ◆ První podproces bude mít hodnotu 43, druhý 43 a hodnota proměnné `lastIdUsed` bude 43.

Překvapivý třetí případ<sup>3</sup> nastane, když si oba podprocesy vlezou do zelí. To se stát může, protože existuje mnoho různých cest, kterými mohou podprocesy zpracovávat stejný řádek kódu v Javě a některé z nich generují chybné výsledky. Kolik takových cest je? Abychom mohli tuto otázku zodpovědět, musíme vědět, co dělá s vygenerovaným bajtovým kódem překladač Just-In-Time a co považuje paměťový model Javy za atomické.

Můžeme hned říci, že při zpracování vygenerovaného bajtového kódu existuje 12 870 různých možných způsobů běhu<sup>4</sup> dvou podprocesů, provádějící kód metody `getNextId`. Kdyby se typ proměnné `getNextId` změnil z `int` na `long`, počet možných způsobů by se zvýšil na 2 704 156. Samozřejmě že většina z nich vygeneruje správné výsledky. Problémem je, že *některé ne*.

## Principy ochrany souběžnosti

Následuje několik principů a technik, které by měly váš systém ochraňovat před problémy souběžného kódu.

### Princip jedné odpovědnosti

Princip jedné odpovědnosti<sup>5</sup> (SRP – Single Responsibility Principle) říká, že daná metoda/třída/komponenta by měla mít jen jeden důvod ke změnám. Souběžný návrh je dost komplikovaný a může být důvodem změn sám o sobě. Je proto zapotřebí, aby byl oddělen od ostatního kódu. Bohužel je běžné, že detaily implementace souběžnosti jsou vloženy přímo do jiného ostrého kódu. Zde je několik záležitostí k zamýšlení:

- ◆ *Souběžný kód má svůj vlastní cyklus vývoje*, změn a ladění.
- ◆ *Souběžný kód má své vlastní problémy*, které jsou jiné a často mnohem komplikovanější, než kód pro nesouběžné úlohy.

3. Viz „Hlubší pohled“ na straně 327.

4. Viz „Počet cest při provádění kódu“ na straně 325.

5. [PPP].

- ◆ Špatně napsaný souběžný kód může selhat z mnoha důvodů a to ho dost komplikuje i bez dalšího břemene okolního aplikačního kódu.

**Doporučení:** *Mějte svůj souběžný kód oddělený od ostatního kódu<sup>6</sup>.*

## Důsledek: omezujte rozsah dat

Jak jsme již viděli, dva podprocesy, které modifikují stejně pole sdíleného objektu, si mohou navzájem překážet a způsobit neočekávané výsledky. Jedním řešením je použít klíčové slovo synchronized, aby se chránila *kritická sekce* kódu, která sídlený objekt používá. Je důležité počet takových kritických sekcí omezit. Čím více je míst, kde se mohou sdílená data aktualizovat, tím spíše je možné, že:

- ◆ Zapomenete zabezpečit jedno nebo několik takových míst – a účinně narušíte veškerý kód, který tato sdílená data modifikuje.
- ◆ Bude zapotřebí dvojnásobného úsilí k zajištění, že vše je pod dohledem (porušení pravidla neopakuje se)<sup>7</sup>.
- ◆ Bude obtížnější určovat zdroj selhání, a to je samo o sobě obtížné.

**Doporučení:** *Používejte důsledně zapouzdření dat; zásadně omezte přístup k jakýmkoliv datům, která by mohla být sdílena.*

## Důsledek: používejte kopie dat

Dobrým způsobem, jak se vyhnout sdílení dat, je v prvé řadě vyhnout se jejich samotnému používání. V některých situacích je možné objekty zkopirovat a používat je jen pro čtení. V jiných případech je možné objekty zkopirovat, v kopiích kumulovat výsledky z různých podprocesů a pak je sloučit do jednoho podprocesu.

Pokud existuje nějaký způsob, jak zabránit sdílení objektů, bude mít výsledný kód mnohem méně problémů. Možná se zabýváte režii na vytváření objektů navíc. Stálo by za námahu vyzkoušet si, zda tento problém opravdu stojí za to. Avšak díky používání kopíí objektů je možné se v kódu vyhnout synchronizaci. Úspora, při které nemusíme používat vnitřní zámky, může vynahradit přídavné režijní náklady na vytváření objektů a uvolňování paměti.

## Důsledek: Podprocesy by měly být co nejméně závislé

Podívejte se na takový zápis kódu, v kterém každý podproces existuje ve svém vlastním světě a žádná data nesdílí s jinými. Každý podproces zpracovává jeden požadavek klienta, veškerá jeho požadovaná data pocházejí z nesdíleného zdroje a jsou uložena v lokálních proměnných. To způsobuje, že se každý podproces chová, jako by byl jediný na světě a jako by nebylo zapotřebí žádné synchronizace.

Například třídy, které byly odvozeny od `HttpServlet`, získávají veškeré své informace jako parametry, jež se předávají metodám `doGet` a `doPost`. To způsobuje, že každý `Servlet` funguje, jako by byl samostatným automatem. Dokud bude kód v tomto programu používat pouze lokální proměnné, není možnost, aby způsoboval synchronizační problémy. Samozřejmě že většina aplikací, které používají servlety, musí někdy sáhnout na sdílené zdroje, jako jsou například databáze.

6. Viz „Příklad klient/server“ na straně 322.

7. [PRAG].

**Doporučení:** Pokuste se rozdělit data na nezávislé podmnožiny, které lze zpracovat v nezávislých podprocesech a pokud možno na různých procesorech.

## Vyznejte se ve své knihovně

Oproti minulým verzím nabízí Java 5 pro vývoj souběžného kódu mnoho vylepšení. Existuje několik věcí, které stojí při psaní podprocesů v Javě 5 za úvahu:

- ◆ Používejte poskytované kolekce, které jsou pro souběžný kód bezpečné.
- ◆ Pro provádění nesouvisejících úloh používejte exekuční rámec (executor framework).
- ◆ Kdykoliv je to možné, používejte řešení, která neblokují.
- ◆ Existuje několik knihovních tříd, které nejsou z hlediska souběžného kódu bezpečné.

### Kolekce, které jsou z hlediska souběžného kódu bezpečné

Když byla Java novým jazykem, napsal Doug Lea originální knihu<sup>8</sup> *Souběžné programování v Javě*. Spolu s touto knihou vyvinul několik kolekcí bezpečných pro souběžný kód, které se později staly součástí JDK (Java Development Kit) v balíčku `java.util.concurrent`. Kolekce v tomto balíčku jsou pro operace s více podprocesy bezpečné a mají dobrý výkon. Implementace `ConcurrentHashMap` pracuje v praxi téměř vždy lépe než `HashMap`. Umožňuje i simultánní souběžné čtení nebo zápisy a obsahuje metody, které podporují společné kombinované operace, jež z hlediska souběžného kódu jinak bezpečné nejsou. Je-li Java 5 prostředím, v němž svůj systém nasadíte, začněte s kolekcí `ConcurrentHashMap`.

Existuje několik dalších druhů tříd, které podporují pokročilý návrh souběžného kódu. Zde je několik příkladů:

<b>ReentrantLock</b>	– zámek, který lze získat v jedné metodě a v druhé uvolnit.
<b>Semaphore</b>	– implementace klasického semaforu, zámku s čitačem.
<b>CountDownLatch</b>	– zámek, který čeká na určitý počet událostí, než uvolní všechny podprocesy, jež na něj čekají. To umožňuje všem podprocesům, aby měly dobrou šanci startovat ve stejný čas.

**Doporučení:** Projděte si třídy, které máte k dispozici. V případě Javy se seznamte s balíčky `java.util.concurrent`, `java.util.concurrent.atomic` a `java.util.concurrent.locks`.

## Poznejte své běhové modely

Existuje několik různých způsobů, jak v souběžné aplikaci oddělovat funkčnosti. Abychom si je mohli probrat, musíme rozumět některým základním definicím.

Vázání zdroje	Zdroje stálé velikosti nebo stálého počtu, používané v souběžném prostředí. Například databázová připojení nebo vyrovnávací paměti pro čtení a zápis.
Vzájemné vyloučení	Jen jeden podproses může najednou přistupovat ke sdíleným datům nebo sdíleným zdrojům.

8. [Lea99].

Vyhledování	Jeden podproces nebo skupina podprocesů má pozastaveno zpracování na velmi dlouhou dobu nebo navždy. Budeme-li například zpracovávat jako první rychlé podprocesy, mohou být delší podprocesy odstaveny, pokud rychlé podprocesy nebudou ukončeny.
Zablokování (uváznutí, deadlock)	Dva nebo několik podprocesů na sebe vzájemně čeká, aby mohly dokončit svou činnost. Každý podproces má k dispozici zdroj, který potřebuje také ten druhý a bez něhož žádný podproces nemůže svou činnost ukončit.
Aktivní zablokování	Zpracovávané podprocesy se snaží pokračovat, ale zjišťují, že jim ten druhý „stojí v cestě“. Díky rezonanci se podprocesy snaží pokračovat dále, ale nedáří se jim to po nepřiměřeně dlouhou dobou nebo navždy.

S těmito definicemi pak můžeme probírat různé běhové modely, které se používají v souběžném programování.

## Producent-spotřebitel<sup>9</sup>

Jeden nebo několik podprocesů-producentů provede nějakou práci a umístí výsledek do vyrovnaných front nebo do fronty. Jeden nebo několik podprocesů-spotřebitelů tuto práci převeze z fronty a dokončí ji. Fronta mezi producenty a spotřebiteli se nazývá **vázaný zdroj**. To znamená, že producenti musejí čekat na volný prostor ve frontě, než mohou dále zapisovat, a spotřebiteli musejí čekat, dokud není něco ve frontě ke zpracování. Koordinace mezi producenty a konzumenty přes frontu vyžaduje, aby mezi nimi byla nějaká signalizace. Producent zapisuje do fronty a signalizuje, že fronta již není prázdná. Konzument čte z fronty a signalizuje, že fronta již není plná. Oba podprocesy teoreticky čekají, dokud nedostanou upozornění, že mohou pokračovat.

## Čtenáři-zapisovatelé<sup>10</sup>

Máte-li sdílený zdroj, který primárně slouží jako zdroj informací pro čtenáře, ale jež příležitostně aktualizují zapisovatelé, musíme řešit otázkou výkonu. Kladení důrazu na výkon může způsobit vyčerpání a akumulaci zastaralých informací. Když umožníme aktualizaci, bude to mít dopad na výkon. Nastaví koordinaci procesů-čtenářů, aby nečetli něco, co právě aktualizuje zapisovatel a naopak, je obtížné. Zapisovatelé mají tendenci blokovat mnoho čtenářů po dlouhou dobu, čímž způsobují problémy s výkonom.

Úkolem je vyvážit potřeby jak čtenářů, tak i zapisovatelů, aby podmínky vyhovovaly korektní činnosti, aby bylo dosaženo rozumného výkonu a předcházelo se zastarání dat. Jednoduchá strategie nechá zapisovatele čekat, dokud existují čtenáři, a teprve poté může zapisovatel provést aktualizaci. Na druhé straně, pokud máme časté zápis, které mají navíc přednost, výkon tím utrpí. Řešením je nalezení správného vyvážení a předcházení vzniku problémů při současné aktualizaci.

9. <http://en.wikipedia.org/wiki/Producer-consumer>.

10. [http://en.wikipedia.org/wiki/Readers-writers\\_problem](http://en.wikipedia.org/wiki/Readers-writers_problem).

## Stolující filozofové<sup>11</sup>

Představte si několik filozofů, kteří sedí u kulatého stolu. Po levici každého z nich je vidlička. Uprostřed stolu je mísa plná špaget. Filozofové po celý čas přemýšlejí, dokud nedostanou hlad. Když už jednou dostanou hlad, chopí se vidliček vlevo i vpravo a začnou jíst. Filozof jíst nemůže, dokud nemá obě vidličky. Pokud filozof vlevo nebo vpravo již některou z vidliček používá, musí čekat, dokud tento filozof nedojí a nepoloží vidličku zpátky na stůl. Když některý filozof dojí, položí obě vidličky na stůl a čeká, až dostane znova hlad.

Zaměňte filozofy za podprocesy a vidličky za zdroje a máme podobný problém, společný mnoha podnikovým aplikacím, ve kterých procesy soutěží o zdroje. Pokud nejsou pečlivě navrženy, mohou se takto soutěžící systémy zablokovat, může u nich nastat aktivní uváznutí a výkon i efektivita může být znehodnocena.

Většina problémů se souběžnosti, se kterými se můžete setkat, bude jedním z těchto tří problémů. Prostudujte si tyto algoritmy a napište vlastní řešení, abyste na ně byli lépe připraveni, když se s nimi setkáte.

**Doporučení:** Prostudujte si tyto základní algoritmy a snažte se pochopit jejich řešení.

## Pozor na závislosti mezi synchronizovanými metodami

Závislosti mezi synchronizovanými metodami mohou způsobit v souběžném kódu jemné chyby. Jazyk Java používá pojem (a klíčové slovo) `synchronized` znamenající ochranu individuální metody. Avšak pokud máme ve stejně sdílené třídě několik synchronizovaných metod, je nebezpečí, že váš systém není dobře napsán<sup>12</sup>.

**Doporučení:** Nepoužívejte u sdíleného objektu více než jednu metodu.

Stane se, že budete muset použít více než jednu metodu sdíleného objektu. Když takový případ nastane, existují tři způsoby, jak napsat kód správně:

- ◆ **Zamykání vycházející z klienta** – před voláním první metody klient uzamyká server a zajistí, že rozsah uzamykání zahrnuje i kód volající poslední metodu.
- ◆ **Zamykání vycházející ze serveru** – vytvořte na serveru metodu, která uzamyká server, zavolejte všechny metody a pak vše uvolněte. Poté může klient volat novou metodu.
- ◆ **Modifikovaný server** – vytvořte prostředníka, který provede zamykání. Je to příklad zamykání vycházejícího ze serveru, kdy nelze modifikovat originální server.

## Mějte synchronizované sekce malé

Klíčové slovo `synchronized` zamyká. Pro všechny sekce kódu pod stejným zámkem je zajištěno, že je bude v kterémkoliv okamžiku provádět jen jeden podproces. Takže nechceme kód těmito příkazy

11. [http://en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](http://en.wikipedia.org/wiki/Dining_philosophers_problem).

12. Viz „Závislost mezi metodami může souběžný kód porušit“ na straně 333.

zaneřádit. Na druhé straně kritické sekce<sup>13</sup> musejí být pod dozorem. Takže chceme, aby návrh kódu byl s co nejmenším počtem kritických sekcí.

Některí naivní programátoři se toho pokouší dosáhnout tím, že píšou kritické sekce velmi dlouhé. Avšak s rozšiřováním synchronizace mimo nejmenší kritickou sekci vzrůstá i soutěž o zdroje a snižuje se výkon<sup>14</sup>.

**Doporučení:** Mějte synchronizované sekce co nejmenší.

## Je obtížné napsat korektní kód pro vypínání

Vytvářet systém, který má neustále běžet, je něco jiného než vytvářet něco, co má fungovat jen chvíli a správně se vypne.

Správné vypnutí může být obtížnou záležitostí. Obecné problémy zahrnují zablokování<sup>15</sup>, díky podprocesům, které očekávají signál k pokračování, jenž ale nikdy nepřijde.

Představte si například systém s nadřazeným procesem, který vytvoří několik podřízených podprocesů a pak čeká na jejich ukončení, aby mohl uvolnit své zdroje a vypnout se. Co když bude některý z podřízených podprocesů zablokován? Nadřazený proces bude čekat donekonečna a systém se nikdy nevypne.

Nebo se podívejte na podobný systém, který dostal pokyn, aby se vypnul. Nadřazený proces sdělí všem podřízeným procesům, aby zanechaly svých úloh a ukončily činnost. Ale co když dva z podřízených procesů fungují jako pář producent-spotřebitel? Předpokládejme, že producent obdrží od nadřazeného procesu signál a rychle svou činnost ukončí. Spotřebitel může očekávat zprávu od producenta a může být zablokován ve stavu, kdy nemůže přijmout signál k vypnutí. Může uvíznout, zatímco čeká na producenta, nikdy neskončit a tak zabránit nadřazenému procesu, aby skončil také.

Podobné situace nejsou neobvyklé. Proto pokud musíte psát souběžný kód, který zahrnuje správné vypínání, předpokládejte, že na jeho správném návrhu strávíte hodně času.

**Doporučení:** Promýšlejte zavážně vypínání a uvedte ho brzy do provozu. Bude to trvat déle, než byste očekávali. Přezkoumejte existující algoritmy, protože to bude patrně obtížnější, než si myslíte.

## Testování kódu podprocesů

Dokazovat, že je kód bezchybný, je nepraktické. Testování bezchybnost negarantuje, dobré testování však může riziko minimalizovat. To je vše pravda pro procesy s jedním podprocesem. Jakmile máme dva nebo několik podprocesů, které využívají stejný kód a sdílejí data, věci se podstatně komplikují.

**Doporučení:** Pište testy, které mohou odhalit problémy, a pak je často spouštějte s různými programovými a systémovými konfiguracemi a různou zátěží. Jestliže někdy selžou, nalezněte místo selhání. Neignorujte selhání jen proto, že během následujících testů k chybě nedošlo.

13. Kritická sekce je jakákolič část kódu, která musí být chráněna před simultáním použitím, aby program fungoval správně.

14. Viz „Zvyšování propustnosti“ na straně 336.

15. Viz „Zablokování“ na straně 338.

To je celkem dost věcí, které je nutné vzít v úvahu. Zde je několik dalších detailnějších doporučení:

- ◆ Berte nejasná selhání jako budoucí možné problémy podprocesů.
- ◆ Uveďte nejdříve do provozu kód bez podprocesů.
- ◆ Vytvářejte souběžný kód jako zásuvný modul.
- ◆ Vytvořte kód s podprocesy s možností nastavení.
- ◆ Spouštějte více procesů, než máte procesorů.
- ◆ Spouštějte kód na různých platformách.
- ◆ Upravte kód tak, aby vyzkoušel a navodil selhání.

## Berte nejasná selhání jako budoucí možné problémy podprocesů

Kód podprocesů může zavinit selhání i tam, kde „to prostě nemůže selhat“. Většina vývojářů (včetně autora) nemá intuitivní cit pro to, jak reagují podprocesy s ostatním kódem. Chyb v kódu podprocesů mohou ukázat své symptomy jednou za tisíc nebo milion spuštění. Pokusy o zopakování mohou být frustrující. To často vede vývojáře k tomu, aby selhání ignorovali jako důsledek kosmických paprsků, hardwarové závady nebo něco „jednorázového“. Nejlepší bude předpokládat, že nic jednorázového prostě neexistuje. Čím déle jsou tyto „výjimky“ ignorovány, tím jistější je, že kód vytváříte na základě potenciálně špatného přístupu.

**Doporučení:** Neignorujte systémová selhání jako výjimky.

## Uveďte nejdříve do provozu kód bez podprocesů

To se zdá být samozřejmé, ale nebude na závadu, když to připomeneme. Ověřte si, že kód funguje samostatně i mimo podproces. Obecně to znamená vytvářet objekty POJO, které jsou volány z podprocesů. Objekty POJO se o podprocesy nestarají a mohou být testovány mimo prostředí s podprocesy. Čím větší část vašeho systému můžete umístit do objektů POJO, tím lépe.

**Doporučení:** Nepokoušejte se hledat běžné chyby zároveň s chybami souběžného kódu. Ujistěte se, že váš kód funguje i mimo podprocesy.

## Vytvářejte souběžný kód jako zásuvný modul

Pište kód podporující souběžnost tak, aby jej bylo možné spustit v několika konfiguracích:

- ◆ Jeden podproces, několik podprocesů se změnami během provádění.
- ◆ Interakce souběžného kódu s reálným nebo náhradním objektem.
- ◆ Spouštějte kód s náhradními objekty, které jsou rychlé, pomalé, proměnlivé.
- ◆ Konfigurujte testy tak, aby mohly běžet opakováně.

**Doporučení:** Vytvořte svůj souběžný kód zásuvný, aby mohl fungovat v různých konfiguracích.

## Vytvořte souběžný kód nastavitelný

Mít správný počet podprocesů vyžaduje v typickém případě metodu pokusu a omylu. Hned zpočátku hledejte způsoby, jak načasovat výkon svého systému za použití různých konfigurací. Počet pod-

procesů by měl být jednoduše nastavitelný. Zvažte, zda by nebylo možné je měnit i za chodu systému. Zvažte, zda by nebylo možné, aby se počet podprocesů nastavoval na základě výkonu a využití systému automaticky.

## Spouštějte více procesů, než máte procesorů

K tomu dochází, když systém přepíná mezi jednotlivými úlohami. Pro podporu přepínání úloh spouštějte více podprocesů, než máte procesory nebo jader. Čím častěji budete přepínat úlohy, tím pravděpodobněji narazíte na kód, který nemá deklarovanou kritickou sekci nebo způsobuje uváznutí.

## Spouštějte kód na různých platformách

V polovině roku 2007 jsme zavedli kurs souběžného programování. Kurs byl primárně založen na OS X. Prezentace byla ve třídě prováděna za použití Windows XP, které fungovaly na bázi VM. Ukázkové testy neselhávaly tak často v prostředí XP, jako v prostředí OS X.

Ve všech těchto případech jsme věděli, že testovaný kód není správný. To jen podpořilo fakt, že různé operační systémy mají různé způsoby jak zacházet s podprocesy a každý z nich má vliv na provoz kódu.

Souběžný kód se chová jinak v různých prostředích<sup>16</sup>. Měli byste spouštět testy v každém prostředí, kde budete potenciálně chtít aplikaci nainstalovat.

**Doporučení:** Spouštějte svůj souběžný kód na všech cílových platformách včas a často.

## Upřavte kód tak, aby vyzkoušel a navodil selhání

Je běžné, že v souběžném kódu se chyby dokážou skrýt. Často se během normálního provozu schovají a jednoduché testy je mnohdy neodhalí. Mohou se objevit jednou za několik hodin, dnů nebo týdnů!

Důvodem, proč jsou chyby v souběžném kódu málo frekventované, sporadicke a obtížné se reprodukují, je, že jen několik z mnoha tisíců možných cest vedoucích přes citlivou sekci může způsobit selhání. Takže pravděpodobnost, že kód půjde cestou, která selže, je překvapivě malá. To ztěžuje detekci a ladění.

Jak byste mohli své šance na zachycení tak ojedinělých událostí zvýšit? Můžete svůj kód upravit a přinutit ho, aby běžel v jiném uspořádání tak, že budete volat metody `Object.wait()`, `Object.yield()`, `Object.sleep()` a `Object.priority()`.

Každá z nich může pořadí provádění kódu ovlivnit, a tudíž může zvýšit naděje na nalezení chyby. Je lepší, když chybnej kód selže brzy a tak často, jak je to jen možné. Pro úpravu kódu existují dvě možnosti:

- ◆ Ruční kódování.
- ◆ Automatizované.

16. Věděli jste, že model podprocesů v Java nezaručuje jejich preemptivní zpracování? Moderní operační systémy jej podporují, takže je dostáváte „zdarma“. Přesto není v prostředí JVM zaručeno.

## Ruční kódování

Do kódu můžete ručně vložit metody `wait()`, `sleep()`, `yield()` a `priority()`. Pro testování zvláště ozechavého kódů to může být to pravé ořechové.

Zde je příklad:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        String url = urlGenerator.next();
        Thread.yield(); // inserted for testing.
        updateHasNext();
        return url;
    }
    return null;
}
```

Vložení volání metody `yield()` změní cestu, kterou kód běží, a může způsobit selhání tam, kde kód dříve neselhával. Jestliže kód selže, nebude to proto, že jste přidali volání metody `yield()`<sup>17</sup>. Pravděpodobnější bude, že váš kód nebyl správný a tento úkon závadu odhalil.

V tomto přístupu existuje mnoho problémů:

- ◆ Pro tento zásah musíte nalézt vhodná místa ručně.
- ◆ Jak víte, kam vložit volání a jakého má být druhu?
- ◆ Jestliže tento kód zanecháte v ostrém prostředí, můžete jeho chod zbytečně zpomalit.
- ◆ Je to jako výstrel z brokovnice. Můžete nalézt chyby a taky nemusíte. V praxi platí, že okolnosti vám příliš nefandí.

To je potřeba provést během testování, ale ne v ostrém kódu. Mezi jednotlivými chody potřebujeme také kombinovat různé konfigurace, což celkově zvyšuje šance, že nalezneme chyby.

Je jasné, že pokud systém rozdělíme na několik objektů POJO, které nemají ponětí o podprocesech a třídách, jež tyto podprocesy řídí, bude hledání vhodných míst pro úpravu kódu jednodušší. Kromě toho bychom mohli vytvořit mnoho různých zátěžových testů, které volají objekty POJO za různých režimů volání metod `sleep`, `yield` atd.

## Automatizované kódování

Pro programovou úpravu svého kódu můžete použít nástroje, jako je Aspect-Oriented Framework, CGLIB nebo ASM. Například byste mohli použít třídu s jedinou metodou:

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

Na různá místa svého kódu můžete přidávat volání:

<sup>17</sup>. To není přesně ten případ. Protože prostředí JVM nezaručuje preemptivní zpracování podprocesů, nějaký algoritmus by mohl v operačních systémech, který podprocesy preemptivně nezpracovává, fungovat vždy. Z jiných důvodů je možná i opačná situace.

```
public synchronized String nextUrlOrNull() {  
    if(hasNext()) {  
        ThreadJigglePoint.jiggle();  
        String url = urlGenerator.next();  
        ThreadJigglePoint.jiggle();  
        updateHasNext();  
        ThreadJigglePoint.jiggle();  
        return url;  
    }  
    return null;  
}
```

Nyní můžete použít jednoduchý aspekt, který na základě náhodného výběru buď nedělá nic, nebo je ve stavu spánku, nebo nechá běžet jiný kód.

Nebo si představte, že třída `ThreadJigglePoint` má dvě implementace. V prvním případě implementuje metodu `jiggle`, která nedělá nic a používá se v ostrém kódu. V druhém případě generuje náhodné číslo a vybírá další činnost, což může být spánek, vracení hodnoty nebo prosté selhání. Můžete spustit tisíc náhodných testů a možná, že se vám podaří některé chyby vymýtit. Pokud testy projdou, alespoň můžete říci, že jste se snažili. I když to může znít poněkud zjednodušeně, může to být rozumnější volba místo nějakého rafinovanějšího nástroje.

Existuje nástroj ConTest<sup>18</sup>, vyvinutý v IBM, který dělá něco podobného, ale o něco rafinovaněji.

Hlavní myšlenkou je náhodně zatěžovat kód tak, aby jeho podprocesy běžely v různém pořadí a v různých časech. Kombinace dobře napsaných testů a zátěží může značně zvýšit šanci pro nalezení chyb.

**Doporučení:** Pro nalezení chyb použijte zátěžové strategie.

## Závěr

Opravovat souběžný kód je obtížné. Jednoduchý kód se může stát noční můrou, když do práce se sdílenými daty zamíchá více podprocesů. Jestliže stojíte před úkolem psát souběžný kód, musíte se přísně držet pravidel psaní čistého kódu nebo se dočkáte jemných chyb a vzácných selhání.

Na prvním místě dodržujte princip jedné odpovědnosti. Rozdělte svůj systém na objekty POJO, které oddělují kód, pracující s podprocesy od kódu, jenž s nimi nepracuje. Během testování kódu pracujícího s podprocesy se přesvědčte, že testujete pouze tento kód a nic jiného. To vede k závěru, že by měl byt krátký a úzce zaměřený.

Seznamte se s případnými informačními zdroji problematiky souběžného programování: vícenásobné podprocesy pracující se sdílenými daty nebo se společnými sdílenými zdroji. Případy, které nelze přesně zařadit, jako je čisté vypínání systému nebo ukončení iterace cyklu, mohou být velmi ošemetné.

Seznamte se se svou knihovnou a s jejími základními algoritmy. Prozkoumejte, jak podporují některé funkčnosti knihovny řešení problémů a nakolik jsou tyto funkčnosti podobné uvedeným základním algoritmům.

18. <http://www.alphaworks.ibm.com/tech/contest>.

Seznamte se, jak nacházet oblasti kódu, které je nutné zamknout, a zamkněte je. Nezamykejte ty oblasti, které to nepotřebují. Vyhýbejte se volání jedné zamknuté sekce kódu z jiné. To vyžaduje důkladnou znalost, zda něco je či není sdílené. Mějte co nejmenší počet sdílených objektů i rozsah jejich sdílení. Modifikujte návrhy objektů se sdílenými daty a snažte se takto raději vyhovět zákazníkům, nenuťte je, aby se o to starali sami.

Problémy se budou objevovat nečekaně. Ty, které se neobjeví na začátku, jsou často ignorovány jako jednorázové záležitosti. V typických případech dochází k témtoto takzvaným jednorázovkám během zátěže nebo ve zdánlivě náhodných intervalech. Je proto zapotřebí, aby váš kód s podprocesy pracoval opakováně a permanentně s mnoha konfiguracemi a na mnoha platformách. Testovatelnost, která jde ruku v ruce s dodržováním tří zákonů vývoje řízeného testy, znamená fungovat do určité úrovně jako plug-in, což poskytuje podporu potřebnou pro běh kódu na mnoha jiných konfiguracích.

Vaše šance na nalezení chybného kódu můžete zvýšit, když budete úpravě kódu věnovat dost času. To můžete provést buď ručně, nebo pomocí nějaké automatizované techniky. Věnujte se tomu co nejdříve. Spouštějte váš kód s podprocesy co nejdéle před jeho nasazením do ostré aplikace.

Bude-li váš přístup čistý, vaše naděje na bezchybnost kódu značně stoupne.

## Použitá literatura

[Lea99]: *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.

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

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

# KAPITOLA 14

## Postupné vylepšování

### V této kapitole najdete:

- ◆ Rozbor analyzátoru argumentů příkazového řádku
- ◆ Implementace třídy Args
- ◆ Třída Args: nanečisto
- ◆ Řetězcové argumenty
- ◆ Závěr



# Rozbor analyzátoru argumentů příkazového řádku

V této kapitole rozebíráme postupné vylepšování kódu. Uvidíte modul, který začínal slabně, ale nerozvíjel se. Pak uvidíte jeho refaktorování a vyčistění. Většina z nás musela čas od času analyzovat argumenty příkazového řádku. Pokud k tomu nemáme vhodný nástroj, musíme jednoduše procházet pole řetězců, které se předávají funkci `main` jako parametry. Existuje několik dobrých programů z různých zdrojů, ale žádný z nich nedělá přesně to, co chci. Samozřejmě že jsem se rozhodl si napsat vlastní. Nazval jsem jej `Args`.

Jeho používání je velmi jednoduché. Vytvoříte třídu `Args` se vstupními argumenty. Podívejte se na následující příklad:

## Výpis 14.1. Jednoduché použití 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("Argument error: %s\n", e.errorMessage());
    }
}
```

Vidíte, jak je to jednoduché. Vytvoříme pouze instanci třídy `Args` se dvěma parametry. Tím prvním je formátovací řetězec nebo *schéma*: `"-l, p#, d*"`. Definuje tři argumenty příkazového řádku. Prvním z nich je `-l` a je typu `boolean`. Druhým je `-p` typu `integer`. Třetím z nich je `-d` typu `string`. Druhým parametrem konstruktoru `Args` je pole argumentů příkazového řádku předávaného funkci `main`.

Když konstruktor vrátí hodnotu, aniž by vyvolal výjimku typu `ArgsException`, byl vstupní příkazový řádek úspěšně analyzován a instanci `Args` je možné posílat dotazy. Metody jako `getBoolean`, `getInteger`, a `getString` nám umožňují přistupovat k hodnotám argumentů podle svých jmen.

Pokud vznikne problém, ať už ve formátovacím řetězci nebo v argumentech příkazového řádku, vyvolá se výjimka typu `ArgsException`. V metodě výjimky `errorMessage` můžeme získat adekvátní popis toho, co bylo špatně.

# Implementace třídy Args

Na výpisu 14.2 je implementace třídy `Args`. Pročtěte si ji, prosím, velmi pečlivě. Věnoval jsem jejímu stylu a struktuře velmi mnoho práce a doufám, že stojí za to si ji projít.

## Výpis 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.nextInt();
}
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));
}
}
}

```

Všimněte si, že tento kód můžete číst odhora dolů bez nějakého skákání vpřed a vzad. Jedinou věcí, kterou budete zřejmě muset nastudovat dopředu, je definice rozhraní ArgumentMarshaler, již jsem úmyslně vynechal. Po pečlivém přečtení tohoto kódu byste měli pochopit, co dělá rozhraní ArgumentMarshaler a co dělají typy od něj odvozené. Ukážu vám několik z nich, abyste to věděli. (Výpis 14.3 až 14.6).

#### **Výpis 14.3. ArgumentMarshaler.java**

```

public interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
}

```

**Výpis 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;  
    }  
}
```

**Výpis 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 "";  
    }  
}
```

**Výpis 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;
}
}
```

Ostatní odvozená rozhraní tento vzor pro hodnoty typu `double` a `String` prostě replikují a v této kapitole je nepotřebujeme. Ponechávám vám je jako cvičení. Jedna věc by vás mohla potrápit: definice konstant kódu pro ošetření chyb. Ty jsou ve třídě `ArgsException` (výpis 14.7).

#### Výpis 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: Should not get here.";
case UNEXPECTED_ARGUMENT:
    return String.format("Argument -%c unexpected.", errorArgumentId);
case MISSING_STRING:
    return String.format("Could not find string parameter for -%c.", errorArgumentId);
case INVALID_INTEGER:
    return String.format("Argument -%c expects an integer but was '%s'.",
        errorArgumentId, errorParameter);
case MISSING_INTEGER:
    return String.format("Could not find integer parameter for -%c.", errorArgumentId);
case INVALID_DOUBLE:
    return String.format("Argument -%c expects a double but was '%s'.",
        errorArgumentId, errorParameter);
case MISSING_DOUBLE:
    return String.format("Could not find double parameter for -%c.", errorArgumentId);
case INVALID_ARGUMENT_NAME:
    return String.format("%c' is not a valid argument name.", errorArgumentId);
case INVALID_ARGUMENT_FORMAT:
    return String.format("%s' is not a valid argument format.", errorParameter);
}
return "";
}
public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE
}
```

Je zajímavé, kolik kódu je zapotřebí k dotvoření detailů tohoto jednoduchého nápadu. Jedním z důvodů je, že používáme zvláště rozvláčného jazyka. Tím, že je Java staticky typovaný jazyk, vyžaduje mnoho slov, aby vyhověla typovému systému. V jazycích, jako je Ruby, Python nebo Smalltalk by tento program byl mnohem menší<sup>1</sup>. Přečtěte si, prosím, výše uvedený kód ještě jednou. Věnujte zvláštní pozornost pojmenování různých objektů, velikosti funkcí a formátování kódu. Jste-li zkušený programátor, možná, že v různých částech budete mít tu a tam nějaké připomínky ke stylu nebo struktuře. Celkově však doufám, že dospějete k závěru, že program je napsán hezký a má čistou strukturu.

Mělo by být například jasné, jak přidat nový typ argumentu, jako je datum nebo komplexní číslo, a že takový úkon by vyžadoval triviální úsilí. Stručně řečeno, chtělo by to prostě nové odvozené rozhraní ArgumentMarshaler – novou funkci getXXX a nový příkaz case ve funkci parseSchemaElement. Zřejmě bychom museli ještě přidat novou funkci ArgsException.ErrorCode a nové chybové hlášení.

1. Nedávno jsem tento modul přepsal do jazyka Ruby. Měl jen jednu sedminu délky a měl o něco lepší strukturu.

## Jak jsem to udělal?

Nechte mě chvíli soustředit se. Tento program jsem nepsal v jeho současné podobně od počátku až do konce. Co je důležitější, neočekávám, že bych byl schopen napsat čisté a elegantní programy na první pokus. Jestliže jsme se během posledních několika desetiletí něčemu naučili, je to zjištění, že programování je více dovedností než vědou. Chcete-li napsat čistý kód, musíte nejdříve napsat kód špinavý a pak jej pročistit.

To by vás nemělo překvapovat. Tohle jsme se naučili na základní škole, když se naši učitelé pokoušeli (obvykle marně), abychom své slohové práce psali nanečisto. Proces, jak říkali, spočíval v tom, že jsme to měli napsat nanečisto, pak druhou verzi, pak několik dalších a pak jsme teprve dospěli k té konečné. Psaní čistých slohových cvičení, jak se nám snažili vysvětlit, je záležitostí postupného vylepšování.

Většina začínajících programátorů (podobně jako většina žáků základní školy) se touto radou neřídí zrovna nejlépe. Věří, že základním cílem je, aby program fungoval. Když už jednou „funguje“, věnují se dalšímu úkolu a ponechávají „fungující“ program v takovém stavu, v jakém jej uvedli do stavu „funkčnosti“. Většina zkušených programátorů ví, že tohle je profesionální sebevražda.

## Třída Args: nanečisto

Výpis 14.8 ukazuje starší verzi třídy Args. „Funguje“. A je chaotická.

### Výpis 14.8. Args.java (první náčrt)

```
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 has invalid format: %s.",
                          elementId, elementTail), 0);
    }
}

private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + " in Args format: " + 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: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.", errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.", errorArgumentId);
    }
}
```

```

        return "";
    }
    private String unexpectedArgumentMessage() {
        StringBuffer message = new StringBuffer("Argument(s) -");
        for (char c : unexpectedArguments) {
            message.append(c);
        }
        message.append(" unexpected.");
        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 {
    }
}

```

Doufám, že vaše první reakce na tento chaos je: „Jsem opravdu rád, že to takhle nenechal!“ Pokud to cítíte, tak si uvědomte, jak se cítí jiní lidé, když vidí váš kód v jeho podobě nanečisto.

„Nanečisto“ je pravděpodobně to nejshovívavější slovo, jakým tento kód můžete označit. Je to zcela jasně rozpracované dílo. Čistý počet instancí proměnných je hrozný. Podivné řetězce, jako je „TILT“, „HashSets“, „TreeSets“ nebo bloky try-catch-catch, to vše přispívá k tomu, že kód vypadá jako hromádka kompostu.

To jsem psát nechtěl. Ve skutečnosti jsem se pokoušel psát kód, který je logický a organizovaný. To můžete se vši pravděpodobností odvodit z mé volby názvů funkcí a proměnných a ze skutečnosti, že struktura programu je hrubá. Ale problém jsem hodil za hlavu.

Chaos vznikal postupně, předchozí verze nebyly tak hrozné. Například výpis 14.9 ukazuje dřívější starší, ve které fungovaly jen argumenty typu Boolean.

**Výpis 14.9. Args.java (jen s argumenty typu 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)) {
        numberArguments++;
        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 numberArguments;
}
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(s) - ");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
}
public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
}
}
```

Ačkoli byste mohli mít k tomuto kódu spoustu námitek, ve skutečnosti není tak špatný. Je kompaktní, jednoduchý a lehce srozumitelný. Avšak v rámci tohoto kódu lze spatřit zřetelná semínka budoucí hromádky kompostu. Je jasné, jak se to změnilo v pozdější chaos. Všimněte si, že pozdější chaotický kód má jen o dva typy argumentů více než tenhle: `String` a `integer`. Přidání pouhých dvou typů argumentů mělo na kód zásadní a negativní vliv. Něco, co by bylo možné rozumně udržovat, se tím změnilo na něco jiného a dalo se očekávat, že to bude prolezlé chybami se spoustou vad na kráse.

Přidal jsem tyto dva typy argumentů postupně. Nejdříve jsem přidal argument typu `String`, který vedl k následujícímu kódu:

**Výpis 14.10. Args.java (s argumenty typu Boolean a 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(
            "Bad character:" + elementId + "in Args format: " + 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("Could not find string parameter for -%c.", errorArgument);
            case OK:
                throw new Exception("TILT: Should not get here.");
        }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    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;
    }
}

```

Můžete vidět, že se nám to pomalu začíná vymykat z rukou. Stále to ještě není tak hrozné, ale zcela určitě zde začíná vystrikovat růžky chaos. Je to hromádka, ale ještě to není zcela ten správný humus. K tomu je zapotřebí přidat typ argumentu `integer`, aby hromádka začala fermentovat a hnít.

## Tak jsem se zastavil

Měl jsem přidat ještě alespoň dva typy argumentů a mohu říci, že by to dopadlo ještě hůře. Kdybych se hnul jako buldozer vpřed a do kódu je zavedl, zanechal bych po sobě chaos, který by již nebylo možné dát do pořádku. Pokud bychom chtěli tuto strukturu někdy udržovat, čas na její opravu nastal právě teď.

Tak jsem přestal přidávat vlastnosti a začal jsem refaktorovat. Protože jsem právě přidal argumenty typů `String` a `integer`, věděl jsem, že pro každý z těchto argumentů bude nutné přidat na třech hlavních místech nový kód. Za prvé, každý argument vyžaduje nějaký způsob, jak analyzovat jeho schéma, aby bylo možno pro tento typ vybrat objekt typu `HashMap`. Dále, každý typ argumentů bylo třeba analyzovat v příkazových řádcích a konvertovat na jeho správný typ. A nakonec, každý typ argumentů potřeboval metodu `getXXX`, aby jej bylo možné vrátit volajícímu objektu jako hodnotu korektního typu. Mnoho různých typů a všechny s podobnými metodami – to mně zní jako třída. A tak se zrodila myšlenka třídy `ArgumentMarshaler`.

## O postupných změnách

Jedním z nejjistějších způsobů, jak poničit program, je provádět masivní změny jeho struktury ve jménu jeho vylepšování. Některé programy se z podobných „zlepšení“ nikdy nevpamatují. Problém je v tom, že je velmi obtížné, aby program pracoval stejným způsobem, jako před „zlepšením“.

Abychom se tomu vyhnuli, používám pravidla vývoje řízeného testy (TDD). Jednou z ústředních pouček tohoto přístupu je, že systém musí fungovat po celou dobu. Jinými slovy, za použití TDD nesmím v systému provést žádnou změnu, která by jej narušila. Po každé změně, kterou provedu, musí systém pracovat stejně jako dříve.

Abych toho dosáhl, potřebuji sadu automatizovaných testů, které mohu spustit v kterémkoli okamžiku a jenž verifikuje, zda je funkčnost systému beze změn. Pro třídu `Args` jsem vytvořil sadu jednotkových a akceptačních testů, když jsem vytvářel hromádku kompostu. Jednotkové testy byly napsány v Javě a k jejich správě jsem použil testovací šablonu `JUnit`. Akceptační testy byly napsány jako stránky wiki pomocí nástroje `FitNesse`. Tyto testy jsem mohl spouštět kdykoliv, a pokud prošly, měl jsem jistotu, že systém pracuje tak, jak má.

Tak jsem pokračoval a prováděl velké množství velmi malých změn. Každá změna posunula strukturu systému směrem ke koncepci třídy `ArgumentMarshaler`. Přitom po každé změně systém fungoval. První změna, kterou jsem provedl, spočívala v tom, že jsem přidal na konec hromádky humusu kostru třídy `ArgumentMarshaler` (výpis 14.11).

#### Výpis 14.11. `ArgumentMarshaler` připojený k `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 {}  
}
```

Bыло ясно, что с этим ничего не покажется. Так я добавил ту самую возможность изменения, о которой говорилось в предыдущем разделе. Изменил я объект типа `HashMap` на объекты типа `Boolean`, чтобы я мог использовать класс `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =  
    new HashMap<Character, ArgumentMarshaler>();
```

To спасло ошибки в нескольких командах, которые я быстро исправил.

```
...  
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());  
}
```

Všimněte si, že tyto změny jsou přesně v těch částech, o kterých jsem se zmiňoval již dříve: v metodách parse, set, a get určených pro jednotlivé typy argumentů. I když byla tato změna malá, některé testy začaly bohužel selhávat. Když se podíváte pečlivě na metodu getBoolean, uvidíte, že když ji zavoláte s parametrem 'y' a argument y neexistuje, vrátí metoda booleanArgs.get('y') hodnotu null a funkce vyvolá výjimku typu NullPointerException. Použili jsme funkci falseIfNull, aby nás před tímto důsledkem pojistila, ale změna, kterou jsem provedl, způsobila, že se tato funkce stala zbytečnou.

Inkrementální postup vyžadoval, abych tohle rychle opravil, dříve než se pustím do jakýchkoliv dalších změn. Oprava nebyla nijak obtížná. Musel jsem pouze přesunout kontrolu hodnoty null. Už jsem nemusel kontrolovat vracenou booleovou hodnotu na null. Byla to třída ArgumentMarshaler.

Nejdříve jsem odstranil volání metody falseIfNull ve funkci getBoolean. Nyní nebyla k ničemu, tak jsem ji kompletně odstranil. Testy selhávaly stejným způsobem i nadále, takže jsem si byl jist, že jsem žádné nové chyby nezavedl.

```
public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
}
```

V dalším kroku jsem rozdělil funkci na dva řádky a třídu ArgumentMarshaler jsem vložil do samostatné proměnné jménem argumentMarshaler. Netoužil jsem po takovém dlouhém jméně. Bylo naprostě nadbytečné a vnášelo do funkce zmatek. Tak jsem ho zkrátil na am [Jm5].

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am.getBoolean();
}
```

A pak jsem vložil logiku pro detekci hodnoty null.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && am.getBoolean();
}
```

## Řetězcové argumenty

Rozšíření o argumenty typu String bylo podobné přidávání logických argumentů. Musel jsem změnit HashMap a zprovoznit metody parse, set, a get. Další postup by neměl být pro nikoho překvapením. Snad až na to, že jsem veškerou implementaci kódu vkládal do základní třídy, místo abych je rozdělil do odvozených tříd.

```
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;
    }
}
```

Opět platí, že se tyto změny prováděly postupně a testy se prováděly tak dlohu, dokud všechny neprosly. Když test selhal, dbal jsem na to, aby byl v pořádku ještě před zavedením dalších změn.

Nyní byste již měli být schopni zpozorovat můj záměr. Když jsem konečně dostal stávající funkčnost do základní třídy ArgumentMarshaler, začal jsem s převodem této funkčnosti směrem dolů do odvozených tříd. To mi umožní udržovat vše ve funkčním stavu, zatímco tvar programu postupně měním.

Dalším samozřejmým krokem bylo přesunout funkctionalitu pro argumenty typu `int` do třídy `ArgumentMarshaler`. Opět se žádné překvapení nekoná.

```
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;
    }
}
```

Když byla veškerá funkcionality přesunuta do třídy ArgumentMarshaler, začal jsem ji přesouvat do odvozených tříd. Prvním krokem bylo přesunout funkci setBoolean do třídy BooleanArgumentMarshaler a přesvědčit se, že se volá korektně. Vytvořil jsem tedy abstraktní metodu 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);
}

```

Pak jsem implementoval metodu set do třídy BooleanArgumentMarshaler.

```

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
        booleanValue = true;
    }
}

```

A nakonec jsem zaměnil volání metody setBoolean voláním metody set.

```

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
}

```

Všechny testy stále procházely. Protože tato změna způsobila, že metoda set byla převedena do třídy BooleanArgumentMarshaler, odstranil jsem metodu setBoolean ze základní třídy ArgumentMarshaler.

Všimněte si, že abstraktní funkce set přebírá argument typu string, ale implementace ve třídě BooleanArgumentMarshaler jej nepoužívá. Tento argument jsem tam vložil, protože jsem věděl, že metody StringArgumentMarshaler a IntegerArgumentMarshaler ji budou používat. Nasazení funkce get je vždy nepříjemné, protože typ návratové hodnoty musí být typu Object a musí být přetytován na Boolean.

```

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
}

```

Jen abych umožnil přklad, přidal jsem funkci get do třídy ArgumentMarshaler.

```
private abstract class ArgumentMarshaler {
    ...
    public Object get() {
        return null;
    }
}
```

Po překladu tohoto kódu testy samozřejmě selhaly. Aby byly opět v pořádku, stačilo funkci get převést na abstraktní a implementovat ji ve třídě BooleanArgumentMarshaler.

```
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;
    }
}
```

Testy opět prošly. Takže obě funkce, get i set, byly nasazeny ve třídě BooleanArgumentMarshaler! To mi umožnilo odstranit ze třídy ArgumentMarshaler starou funkci getBoolean, přesunout chráněnou proměnnou booleanValue dolů do třídy BooleanArgumentMarshaler a změnit ji na soukromou. Stejné kroky jsem provedl pro řetězce. Nasadil jsem funkce set i get, smazal nepoužité funkce a přemístil proměnné.

```
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;
    }
}
```

Nakonec jsem tento proces zopakoval pro argumenty typu `integer`. To bylo trochu komplikovanější, protože tyto argumenty byly předmětem analýzy a operace mohla vyvolat výjimku. Ale výsledek je lepší, protože vše kolem `NumberFormatException` se mohlo skrýt do třídy `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;
    }
}
```

Samozřejmě že testy procházely i nadále. V dalším kroku jsem se zbavil tří různých map ze začátku algoritmu. Tím se stal systém mnohem generičtějším. Avšak nemohl jsem se jich zbavit pouhým vymazáním, protože by to způsobilo nefunkčnost systému. Místo toho jsem musel přidat nový objekt Map pro třídu ArgumentMarshaler a postupně změnit všechny třídy, aby nahradily tři původní mapy.

```
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);
}

```

Samozřejmě že všechny testy stále procházely. V dalším kroku jsem změnil metodu `isBooleanArg` z tvaru:

```

private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
}

```

do tvaru:

```

private boolean isBooleanArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof BooleanArgumentMarshaler;
}

```

Testy stále procházely. Provedl jsem tedy stejné změny v metodách `isIntArg` a `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 stále procházely. Takže jsem odstranil veškerá zdvojená volání metody `marshalers.get` takto:

```

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;
}

```

V tomto okamžiku již neexistují žádné důvody pro existenci tří metod `isxxxArg`. Proto jsem je vložil jako jednořádkové funkce:

```

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;
}

```

V dalším kroku jsem začal používat mapu `marshalers` ve funkcích `set` a přestal jsem používat ostatní tři mapy. Začal jsem s metodami typu `boolean`.

```

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"); // was: booleanArgs.get(argChar).set("true");
    }
}

```

```
    } catch (ArgsException e) {
    }
}
```

Testy stále procházely, takže jsem stejné změny provedl v metodách pro typy String i Integer. To mi umožnilo integrovat některé špatné části kódu pro obsluhu výjimek do funkce 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();
    }
}
```

Nyní jsem již mohl pomalu odstranit tři staré mapy. Nejdříve jsem potřeboval změnit funkci getBoolean z této podoby:

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
}
```

najinou:

```
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;
}
```

Poslední provedená změna by se mohla jevit jako nečekaná. Proč jsem se rozhodl zabývat se najednou třídou `ClassCastException`? Je to proto, že mám sadu jednotkových testů a samostatnou sadu akceptačních testů, které jsou psané v jazyce FitNesse. Ukázalo se, že testy FitNesse ověřily, že pokud voláte funkci `getBoolean` s jiným než logickým argumentem, obdržíte hodnotu `false`. U jednotkových testů nikoliv. Až dosud jsme používali pouze testy jednotkové<sup>2</sup>. Tato změna mi umožnila odstranit další použití mapy typu `boolean`:

```
private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
}
```

A nyní můžeme smazat mapu 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>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    ...
}
```

Dále jsem stejným způsobem přesunul argumenty typů `String` a `Integer` a trochu jsem pročistil metody pro logické argumenty.

```
private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
```

---

2. Abychom zabránili dalším překvapením tohoto druhu, přidal jsem nový jednotkový test, který volal všechny testy z FitNesse.

```
}

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>();

    ...
}
```

Pak jsem převedl tři metody parse jako vložené jednořádkové funkce, protože toho již mnoho neprováděl:

```
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 has invalid format: %s.", elementId, elementTail), 0);
    }
}
```

Dobrá, znovu se nyní podívejme na celek. Výpis 14.12 ukazuje současnou podobu třídy Args.

**Výpis 14.12. Args.java (Po prvním refaktorování)**

```
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 has invalid format: %s.», elementId, elementTail), 0);
    }
}
private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            «Bad character:» + elementId + «in Args format: » + 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: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.", errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.", errorArgumentId);
    }
    return "";
}
private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");
    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 vší této práci je výsledkem určité zklamání. Struktura je o něco lepší, ale stále máme na začátku kódu všechny ty proměnné, stále máme ty nepříjemné větve pro ošetřování typů v příkazu `setArgument` spolu se všemi těmi funkcemi `set`, které jsou opravdu špatné. Nemluvě o celém zpracování chyb. Před námi je stále ještě mnoho práce.

Opravdubych se rád zbavil větví `case` v příkazu `setArgument` [O23]. V tomto příkazu bych chtěl mít jedno volání metody `ArgumentMarshaler.set`. To znamená, že potřebuji přemístit metody `setIntArg`,

setStringArg a setBooleanArg směrem dolů a zavést je do náležité třídy odvozené od ArgumentMarshaler. Ale je zde problém. Když se podíváte trochu podrobněji na metodu setIntArg, všimnete si, že používá dvě instanční proměnné: args a currentArg. Abych mohl přemístit metodu setIntArg dolů do třídy BooleanArgumentMarshaler, budu muset také předávat proměnné args a currentArgs jako funkční argumenty. To není čisté [F1]. Raději bych předával jeden argument než dva. Naštěstí existuje jednoduché řešení. Můžeme konvertovat pole args na list a funkčím set předat Iterator. Následující úkony mně zabraly deset kroků a po každém z nich následovaly úspěšné testy. Ale ukážu vám jen výsledky. Měli byste na většinu z těchto malých kroků přijít sami.

```
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();
    }
}
}

```

Všechny tyto změny byly jednoduché a všechny testy prošly. Nyní můžeme začít s přesunem funkci set směrem dolů do příslušných odvozených tříd. Nejdříve potřebujeme provést následující změny v metodě 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;
}

```

Tato změna je důležitá, protože se chceme zcela zbavit příkazu `if-else`. To znamená, že potřebujeme ostranit chybovou podmínu.

Nyní můžeme začít přesouvat funkce set. Funkce `setBooleanArg` je triviální, takže s ní začneme jako první. Cílem je změnit funkci `setBooleanArg`, aby byla přesměrována na třídu `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) {
}
}

```

Nevložili jsme toto zpracování výjimek právě sem? V procesu refaktorování je zcela běžnou praxí vložit nějaký text, abyste jej později mohli vyjmout. Velmi malé kroky a potřeba neustálého testování je to, co velmi pomáhá postupu vpřed. Refaktorování se do značné míry podobá Rubikově kostce. Existuje mnoho malých kroků, které jsou nutné pro dosažení velkého cíle. Každý z těchto kroků umožňuje přejít k následujícímu.

Proč jsme předávali argument typu `Iterator`, když ho metoda `setBooleanArg` zcela určitě nepotřebuje? Protože ho budou určitě potřebovat metody `setIntArg` a `setStringArg`? A protože chci všechny tři funkce používat prostřednictvím abstraktní metody ve třídě `ArgumentMarshaler`, potřebuji ho předat metodě `setBooleanArg`.

Takže nyní je metoda `setBooleanArg` k ničemu. Kdyby byla ve třídě `ArgumentMarshaler` funkce `set`, mohli bychom ji volat přímo. Nastal tedy čas tuto funkci vytvořit. Prvním krokem bude přidat novou abstraktní metodu do třídy `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();
}

```

Je zřejmé, že tento postup způsobí chyby ve všech odvozených objektech. Implementujme tedy novou metodu v každém z nich.

```
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;
    }
}
```

Nyní můžeme odstranit metodu 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;
    }
}
```

Všechny testy projdou a funkce set je umístěna do třídy BooleanArgumentMarshaler! Nyní můžeme provést stejné kroky v metodách pro řetězce a celá čísla.

```
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;
}
}
}

```

A přichází rána z milosti: Větve case byly odstraněny! Zásah!

```

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;
    }
}
}

```

Nyní se můžeme zbavit některých špatně napsaných funkcí ve třídě IntegerArgumentMarshaler a trochu je pročistit.

```

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;
}
}

```

Můžeme také změnit **ArgumentMarshaler** na rozhraní.

```

private interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
    Object get();
}

```

Nyní se podívejme, jak je jednoduché přidat do naší struktury nový typ argumentu. Mělo by to vyžadovat jen několik málo změn a tyto změny by měly být izolované. Nejdříve přidáme nové testovací větvě ca se, abychom zkontovali, že argument typu double funguje korektně.

```

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);
}

```

Nyní vyčistíme schéma pro analýzu kódu a přidáme detekci # # pro argumenty 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 has invalid format: %s.", elementId, elementTail), 0);
}

```

V dalším kroku napíšeme třídu 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;
}
}

```

To nás nutí přidat novou metodu ErrorCode.

```

private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE
}

```

A potřebujeme funkci 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;
    }
}

```

A všechny testy procházejí! To bylo docela bezbolestné. Takže teď se zkusme přesvědčit, že kód pro zpracování chyb funguje správně. Následující větev case kontroluje, zda je deklarována chyba v případě, že do argumentu typu ## byl načten řetězec, který nelze analyzovat.

```

public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
        args.errorMessage());
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:

```

```
        return unexpectedArgumentMessage();
    case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
            errorArgumentId);
    case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
            errorArgumentId, errorParameter);
    case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
            errorArgumentId);
    case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
            errorArgumentId, errorParameter);
    case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
            errorArgumentId);
    }
    return "";
}
```

A testy procházejí. Následující test ověruje, že jsme schopni správně detekovat chybějící argument typu double.

```
public void testMissingDouble() throws Exception {
    Args args = new Args("x#H#", new String[]{"-x"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
        args.errorMessage());
}
```

Dalo se očekávat, že tento kód testy projde. Napsali jsme jej prostě proto, aby byl kompletní.

Kód pro ošetření výjimek je dost nepříjemný a nepatrí tak docela do třídy `Args`. Vyvoláváme také výjimku typu `ParseException`, která nám příliš nepatrí. Takže slučme všechny výjimky do jedné třídy `ArgsException` a přemístěme ji do vlastního modulu.

```
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 has invalid format: %s.",
                          elementId,elementTail));
}
private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
        throw new ArgsException(
            "Bad character:" + elementId + "in Args format: " + 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;
    }
}
```

To je pěkné. Jedinou výjimkou, kterou vyvolá třída Args, je ArgsException. Přesunutí této výjimky do samostatného modulu znamená, že do tohoto modulu můžeme přesunout značnou část podpůr-

ného kódu pro ošetřování chyb z modulu Args. Je to přirozené a samozřejmé místo, kam takový kód umístit, a pomůže nám to při dalším čistění modulu Args.

Takže nyní jsme zcela oddělili kód pro ošetřování chyb a výjimek od kódu z modulu Args. (Viz výpis 14.13 až 14.16). Toho jsme dosáhli pomocí asi třiceti malých kroků a po každém z nich jsme spouštěli testy.

#### Výpis 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("Args constructor should have thrown exception");
        } 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("Args constructor should have throws exception");
        } 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", "Forty two"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", 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'), .001);
    }

    public void testInvalidDouble() throws Exception {
        try {
            new Args("x##", new String[] {"-x", "Forty two"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
            assertEquals("Forty two", 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());
        }
    }
}
}

```

**Výpis 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 unexpected.", e.errorMessage());
    }

    public void testMissingStringMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
            'x', null);
        assertEquals("Could not find string parameter for -x.", e.errorMessage());
    }

    public void testInvalidIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                'x', "Forty two");
        assertEquals("Argument -x expects an integer but was 'Forty two'.",
            e.errorMessage());
    }

    public void testMissingIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
        assertEquals("Could not find integer parameter for -x.", e.errorMessage());
    }
}

```

```
}

public void testInvalidDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
        e.errorMessage());
}

public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
}

}
```

**Výpis 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: Should not get here.");
            case UNEXPECTED_ARGUMENT:
                return String.format("Argument -%c unexpected.", errorArgumentId);
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.", errorArgumentId);
            case INVALID_INTEGER:
                return String.format("Argument -%c expects an integer but was '%s'.", errorArgumentId, errorParameter);
            case MISSING_INTEGER:
                return String.format("Could not find integer parameter for -%c.", errorArgumentId);
            case INVALID_DOUBLE:
                return String.format("Argument -%c expects a double but was '%s'.", errorArgumentId, errorParameter);
            case MISSING_DOUBLE:
                return String.format("Could not find double parameter for -%c.", errorArgumentId);
        }
        return "";
    }
    public enum ErrorCode {
        OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
        MISSING_STRING,
        MISSING_INTEGER, INVALID_INTEGER,
        MISSING_DOUBLE, INVALID_DOUBLE
    }
}
}

```

#### Výpis 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);
    }
}
```

Většina změn ve třídě `Args` se týkala mazání. Velké množství kódu bylo přesunuto ze třídy `Args` do `ArgsException`. Překně. Přemístili jsme také všechny třídy `ArgumentMarshaler` do samostatných souborů. Ještě lepší!

Mnoho dobrých softwarových návrhů je prostě otázkou rozložení do sekcí – vytvoření vhodných míst, kam umístit rozdílné druhy kódu. Tato separace zájmových oblastí značně zjednoduší porozumění kódu a jeho údržbu.

Zvláštní význam má metoda `errorMessage` z třídy `ArgsException`. Umístění chybového hlášení do třídy `Args` znamená jasné porušení principu jedné odpovědnosti (SRP – Single Responsibility Principle). Třída `Args` by měla zpracovávat jen argumenty, ne formáty jejich chybových hlášení. Avšak má opravdu smysl umisťovat formátovací kód pro chybová hlášení do třídy `ArgsException`? Upřímně řečeno, jde o kompromis. Uživatelé, kterým by se nelíbilo, že chybová hlášení jsou umístěna ve třídě `ArgsException`, si budou muset napsat svou vlastní třídu. Ale vymoženost mít pro vás předem připravená chybová hlášení není bezvýznamná.

Nyní by již mělo být jasné, že jsme na dosah finálnímu řešení, které se objevilo na začátku této kapitoly. Poslední změny jsem si pro vás nechal jako cvičení.

## Závěr

Nestačí, že kód funguje. Kód, který funguje, má často různé závady. Programátoři, kteří se spokojí s pouhým fungováním kódu, se chovají neprofesionálně. Mohou mít pocit, že nebudou mít čas opravovat strukturu a návrh svého kódu, ale nesouhlasím. Nic nemá tak hluboký a dlouhodobý degradující vliv na vývoj projektu jako špatný kód. Špatný plán je možné přepracovat, špatné požadavky mohou být předefinovány. Špatná dynamika týmu může být napravena. Ale špatný kód hnije a kvasí, až se z něj stane neúprosná přítěž, která stahuje celý tým k zemi. Opakovaně jsem viděl týmy, jak se moří, protože vytvořili ve spěchu maligní bahnísko kódu, které od tohoto okamžiku navždy určovalo jejich osud.

Samozřejmě že špatný kód lze vyčistit. Ale je to velmi drahé. S tím, jak kód degeneruje, moduly se do sebe zaklesávají a vytvářejí mnoho skrytých a spletíčných závislostí. Nalezení a zrušení starých závislostí je dlouhá a náročná práce. Na druhé straně, udržování čistého kódu je relativně jednoduché. Pokud uděláte v nějakém modulu dnes ráno nepořádek, je jednoduché jej vyčistit ještě odpoledne. Jestliže jste udělali nějaký nepořádek před pěti minutami, bude ještě lepší, když jej vyčistíte ihned.

Takže řešením je udržovat váš kód neustále tak čistý a jednoduchý, jak jen to je možné. Nikdy nedopusťte, aby degenerace mohla začít.

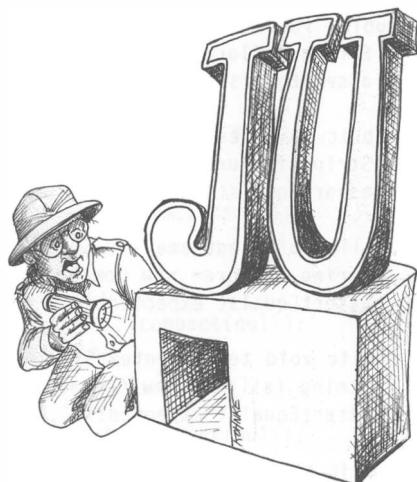


# KAPITOLA 15

## Vnitřní části šablony JUnit

**V této kapitole najdete:**

- ◆ Šablona JUnit
- ◆ Závěr



Testovací šablona JUnit je jedna z nejproslulejších šablon v jazyce Java. Ve srovnání s jinými šablonami má jednoduchou koncepci, precizní definici a elegantní implementaci. Ale jak vypadá kód? V této kapitole se budeme zabývat příkladem, který vznikl v šabloně JUnit.

## Šablona JUnit

JUnit má mnoho autorů, ale začali na ní společně pracovat Kent Beck a Eric Gamma během jednoho letu do Atlanty. Kent chtěl učit Java a Eric chtěl dovědět něco ohledně testovací šablon pro Smalltalk. „Co by mohlo být pro pár tulpasů přirozenější, než když na přecpaném místě vytáhnou laptopy a začnou programovat?“<sup>1</sup> Po třech hodinách práce ve velké výšce napsali základy šablony JUnit.

Modul, který zde uvidíme, je ta chytřejší část kódu, jež napomáhá identifikovat chyby při porovnávání řetězců. Tento modul se nazývá `ComparisonCompactor`. Jsou-li dány dva odlišné řetězce, jako je ABCDE a ABXDE, zobrazí jejich rozdíl a vygeneruje řetězec formátu <...B[X]D...>.

Mohl bych to vysvětlovat ještě dál, ale na modelovém případu to dokážeme lépe. Podívejte se tedy na výpis 15.1 a proniknete mnohem hlouběji do toho, co tento modul vyžaduje. Když jsme se k tomu dostali, zkusme zhodnotit strukturu testů. Mohly by být jednoduší nebo jasnější?

### Výpis 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 expected:<[b]> but was:<[c]>".equals(failure));
    }
    public void testStartSame() {
        String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
        assertEquals("expected:<b[a]> but was:<b[c]>", failure);
    }
    public void testEndSame() {
        String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
        assertEquals("expected:<[a]b> but was:<[c]b>", failure);
    }
    public void testSame() {
        String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
        assertEquals("expected:<ab> but was:<ab>", failure);
    }
    public void testNoContextStartAndEndSame() {
        String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
        assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
    }
    public void testStartAndEndContext() {
        String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
        assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
    }
}
```

1. *JUnit Pocket Guide*, Kent Beck, O'Reilly, 2004, str. 43.

```
public void testStartAndEndContextWithEllipses() {
    String failure=
        new ComparisonCompactor(1, "abcde", "abfde").compact(null);
    assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
}
public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
}
public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
}
public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
}
public void testComparisonErrorOverlapingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
    assertEquals("expected:<...[...]...> but was:<...[b]...>", failure);
}
public void testComparisonErrorOverlapingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
}
public void testComparisonErrorOverlapingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde",
        "abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[...]...>", failure);
}
public void testComparisonErrorOverlapingMatches2Context() {
    String failure=
        new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
}
public void testComparisonErrorWithActualNull() {
    String failure= new ComparisonCompactor(0, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}
public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}
public void testComparisonErrorWithExpectedNull() {
    String failure= new ComparisonCompactor(0, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
}
public void testComparisonErrorWithExpectedNullContext() {
    String failure= new ComparisonCompactor(2, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
}
public void testBug609972() {
```

```

        String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
        assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
    }
}
}

```

Tyto testy jsem použil a za pomoci modulu `ComparisonCompactor` jsem spustil analýzu pokrytí kódu. Kód byl pokryt na sto procent. Každý řádek kódu, každý příkaz `if` nebo cyklus `for` byl témito testy proveden. To mi dává velkou důvěru, že kód funguje správně, a plyne z toho velký obdiv k mistrovské práci autorů.

Kód modulu `ComparisonCompactor` je na výpisu 15.2. Věnujte mu trochu času a podívejte se na něj. Myslím, že kód je dobře rozdelený do sekcí, rozumně expresivní a má jednoduchou strukturu. Až budete hotovi, společně odstraníme jeho chyby.

#### Výpis 15.2. `ComparisonCompactor.java` (původní kód)

```

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);
}
}
```

K tomuto modulu můžete mít lečjaké výhrady. Jsou tam dlouhé výrazy a nějaká zvláštní přičítání jednotek a podobně. Ale celkově je tento modul docela dobrý. Koneckonců, mohl vypadat jako kód z výpisu 15.3.

#### Výpis 15.3. ComparisonCompactor.java (po refaktorování)

```
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;
    }
}

```

Ikdyž autoři zanechali tento modul ve velmi dobrém stavu, *skautské pravidlo*<sup>2</sup> nám říká, že kód máme zanechat čistší, než jaký byl dříve. Lze tedy původní kód z výpisu 15.2 zlepšit?

První věcí, bez které se mohu obejít, je předpona `f` u členských proměnných [Jm6]. V dnešních vývojových prostředích je podobné kódování, určující rozsah platnosti, nadbytečné. Odstraňme je.

```

private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;

```

V dalším kroku máme nezapouzdřené podmínky na začátku funkce `compact` [O28].

---

2. Viz „Skautské pravidlo“ na straně 36.

```
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);
}
```

Tyto podmínky by měly být zapouzdřeny, aby náš záměr byl srozumitelný. Extrahujme tedy metodu, která to objasní.

```
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();
}
```

Označení `this.expected` a `this.actual` ve funkci `compact` nepovažuji za nezbytná. K tomu došlo během změny názvu z `fExpected` na `expected`. Proč v této funkci existují proměnné, které mají stejná jména jako členské proměnné? Nepředstavují něco jiného [Jm4]? Tato jména by měla být jednoznačná.

```
String compactExpected = compactString(expected);
String compactActual = compactString(actual);
```

Negativní pojmy jsou o něco hůře srozumitelné než pojmy pozitivní [O29]. Takže změňme příkaz `if` ze začátku kódu a obraťme smysl jeho podmínek.

```
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();
}
```

Název funkce je divný [Jm7]. I když tato funkce spojuje řetězce, nemusí to provést v případě, že by funkce `canBeCompacted` vrátila hodnotu `false`. Takže pojmenování této funkce jako `compact` skrývá její vedlejší efekt, že kontroluje chybu. Také si všimněte, že funkce vrací nejen sloučené řetězce, ale i formátovanou zprávu. Takže jméno funkce by mělo ve skutečnosti být `formatCompactedComparison`. Pak bude mnohem čitelnější, když se použije společně s funkčním argumentem:

```
public String formatCompactedComparison(String message) {
```

Tělo příkazu `if` je v místě, kde dochází ke skutečnému spojení očekávaného a skutečného řetězce. Tento kód bychom měli extrahovat jako metodu jménem `compactExpectedAndActual`. Avšak po funkci `formatCompactedComparison` chceme, aby prováděla veškerá formátování. Funkce `compact...` by neměla dělat nic jiného než spojování [O30]. Rozdělme ji tedy následovně:

```
...
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);
}
```

Všimněte si, že tento krok si vynutil přeměnu proměnných `compactExpected` a `compactActual` na členské proměnné. Nelibí se mi, že poslední dva řádky nové funkce vracejí proměnné, ale první dva nikoliv. Nepoužívají konzistentní konvenci [O11]. Měli bychom tedy modifikovat metody `findCommonPrefix` a `findCommonSuffix`, aby vracely předpony a přípony.

```
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;
}
```

Měli bychom také změnit názvy členských proměnných, aby byly o něco přesnější [Jm1]. Nakonec jde v obou případech o indexy.

Pečlivá prohlídka funkce `findCommonSuffix` ukáže *skrytou dočasnou vazbu* [O31]. Pokud by program tyto dvě funkce volal v nesprávném pořadí, bylo by ladění obtížné. Abychom tuto dočasnou vazbu ukázali, zavedeme `prefixIndex` jako argument funkce `findCommonSuffix`.

```
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;
}
```

S tím ale nejsem úplně spokojen. Předání argumentu `prefixIndex` je poněkud svévolné [O32]. Zavádí to určité pořadí, ale nijak to nevysvětluje, proč tohle pořadí potřebujeme. Jiný programátor by mohl vrátit kód do původního stavu, protože nic nenasvědčuje tomu, proč je tento parametr opravdu zapotřebí. Udělejme tedy něco jiného.

```
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;
}

```

Vráťme funkce `findCommonPrefix` a `findCommonSuffix` tam, kde byly předtím, změníme jméno funkce `findCommonSuffix` na `findCommonPrefixAndSuffix` a necháme ji volat funkci `findCommonPrefix` před jakoukoliv další činností. To nastolí časovou souslednost těchto dvou funkcí mnohem pozoruhodnějším způsobem než předchozí řešení. Ukazuje se také, jak je funkce `findCommonPrefixAndSuffix` špatná. Nyní ji vycistěme.

```

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;
}

```

To je mnohem lepší. Ukazuje se, že proměnná `suffixIndex` je opravdu délka přípony a není dobré pojmenovaná. Totéž platí pro proměnnou `prefixIndex`, i když v tomto případě „index“ a „délka“ jsou synonyma. I tak je lepší používat termín „délka“. Problém je v tom, že se délka neodvíjí od nuly, ale od jedničky, a nejde tudíž o skutečnou délku. To je také důvodem existence oněch přičítání jedné ve funkci `computeCommonSuffix` [O33]. Opravme to. Výsledek zobrazuje výpis 15.4.

#### **Výpis 15.4. ComparisonCompactor.java (mezikód)**

```

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 : ""));
}
```

Vyměnili jsme přičítání jedné ve funkci `computeCommonSuffix` za odpočet jedné ve funkci `charFromEnd`, kde to má opravdu smysl, a za dvojici operátorů `<=` ve funkci `suffixOverlapsPrefix`, kde je to rovněž odůvodněné. To nám umožnilo změnit název `suffixIndex` na `suffixLength`, což značně zlepšuje čitelnost kódu.

Je tam však jeden problém. Když jsem odstraňoval přičítání jedné, všiml jsem si v metodě `compactString` následujícího rádku:

```
if (suffixLength > 0)
```

Podívejte se na výpis 15.4. Protože je nyní proměnná `suffixLength` o jedničku menší než byla, máme důvod změnit operátor `>` na `>=`. Ale to nedává žádný smysl. Kód je správný *nyní!* Což znamená, že kód byl špatně již dříve a že šlo pravděpodobně o chybu v programu. Možná, že ne tak docela. Po

další analýze zjistíme, že příkaz `if` již neumožnuje přidávat příponu nulové délky. Před touto změnou nebyl příkaz `if` funkční, protože proměnná `suffixIndex` nemohla být nikdy menší než jedna!

To vznáší otazník na oba příkazy `if` ve funkci `compactString`! Zdá se, že bychom mohli oba příkazy odstranit. Zakomentujme je tedy a spusťme testy. Ty prošly! Proveďme tedy restrukturalizaci funkce `compactString`, abychom odstranili nadbytečné příkazy `if`, a funkci zjednodušme [O9].

```
private String compactString(String source) {
    return
        computeCommonPrefix() +
        DELTA_START +
        source.substring(prefixLength, source.length() - suffixLength) +
        DELTA_END +
        computeCommonSuffix();
}
```

To je mnohem lepší! Nyní vidíme, že funkce `compactString` pouze spojuje dohromady fragmenty. Tohle můžeme pravděpodobně vyjádřit jasněji. Je zde opravdu mnoho drobné práce k čistění, do které se můžeme pustit. Ale než bych vás vláčel všemi ostatními změnami, raději vám ukážu výsledek ve výpisu 15.5.

#### Výpis 15.5. ComparisonCompactor.java (konečná verze)

```
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;
}

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 : "");
}
}
```

To už je docela hezké. Modul je separován do několika analytických funkcí a několika syntetických funkcí. Jsou topologicky setříděny tak, že se definice každé funkce objevuje těsně po jejím použití. Všechny analytické funkce jsou uvedené jako první a všechny syntetické funkce jako poslední.

Při bližším pohledu si všimnete, že jsem změnil několik rozhodnutí, která jsem v této kapitole učinil již dříve. Zavedl jsem například jako jednořádkové funkce několik extrahovaných funkcí do metody `formatCompactedComparison` a změnil smysl výrazu `shouldNotBeCompacted`.

To je typické. Jedno refaktorování vede často k jinému, které vede k odstranění kroků toho prvního. Refaktorování je iterativní proces plný pokusů a omylů a nevyhnutelně konvergující k něčemu, o čem se můžeme domnívat, že je to práce profesionála.

## Závěr

Skauskému pravidlu jsme tedy vyhověli. Zanechali jsme tento modul o něco čistší, než byl dříve. Neže by předtím čistý nebyl. Autoři s ním provedli výbornou práci. Ale žádný modul není imunní vůči zlepšování a každý z nás zodpovídá za to, aby zanechal kód ve stavu o něco lepším, než byl.

# KAPITOLA 16

## Refaktorování třídy SerialDate

**V této kapitole najdete:**

- ◆ Nejdřív ať to funguje
- ◆ Pak to sprav
- ◆ Závěr
- ◆ Použitá literatura



Podíváte-li se na stránku <http://www.jfree.org/jcommon/index.php>, naleznete tam knihovnu JCommon. Někde uvnitř této knihovny je balíček jménem `org.jfree.date`. Je v něm třída jménem `SerialDate`. Na tuhlu třídu se podíváme.

Autorem této třídy je David Gilbert. David je zřejmě zkušený a kompetentní programátor. Jak ještě uvidíme, ukazuje prostřednictvím svého kódu značný stupeň profesionality a ukázněnosti. Ze všech hledisek jeho smyslu a účelnosti je tento kód „dobrým kódem“. A já se ho právě chystám rozpitvat na kousky.

Tím nemíním nic zlého. Ani si nemyslím, že bych byl o tolik lepší, než je David, abych si činil nároky na vynášení soudů o jeho kódu. Opravdu ne. Kdybyste nalezli nějaký můj kód, jsem si jist, že byste v něm našli spoustu věcí, které by mohly být terčem kritiky.

Ne, nejdále se o akt nevraživosti nebo arrogance. To, co chci udělat, není nic jiného, než profesionální recenze. Je to něco, co by nikomu z nás nemělo vadit. A je to něco, co bychom měli přivítat, pokud to někdo udělá s naším kódem. Jen za pomoci podobných kritik se něčemu můžeme naučit. Lékaři to dělají. Piloti to dělají. Právníci to dělají. A my programátoři se to potřebujeme naučit také.

Ještě něco o Davidu Gilbertovi: David je něco více, než jen dobrý programátor. David má odvahu a dobrou vůli, aby nabídl svůj kód komunitě ve velkém a zdarma. Zpřístupnil ho veřejnosti, aby se všichni mohli podívat, a vyzval veřejnost, aby kód používala a prozkoumávala. To byl dobrý krok!

Třída `SerialDate` (výpis B.1, str. 352) reprezentuje v Javě datum. Proč máme v Javě třídu, která reprezentuje datum, když tam jsou už třídy `java.util.Date`, `java.util.Calendar` a jiné? Autor napsal tuto třídu jako reakci na potíže, které jsem často pocitoval sám. Dobře to vysvětluje poznámka v úvodu dokumentace vyrobené v Javadoc (rádek 67). O jeho záměru můžeme diskutovat, ale já jsem se tímto tématem určitě zabýval již dříve a vítám třídu, která pracuje s datem a ne s časem.

## Nejdřív ať to funguje

Třída `SerialDateTests` (výpis B.2, str. 372) obsahuje nějaké jednotkové testy. Všechny testy provědou. Rychlá inspekce testů bohužel odhalí, že netestují vše [T1]. Například hledání „Find Usages“ metody `MonthCodeToQuarter` (rádek 334) ukazuje, že se metoda nepoužívá [F4]. Tudiž jednotkové testy ji netestují.

Tak jsem nastartoval program Clover, abych se podíval, co jednotkové testy pokrývají a co ne. Program ohlásil, že jednotkové testy provedly jen 91 z celkového počtu 185 proveditelných řádků, které se nacházely ve třídě `SerialDate` (padesát procent) [T2]. Mapa pokrytí vypadala jako sešívání příkrývka s velkými úseky hlušiny neprovedeného kódu po celé třídě.

Mým cílem bylo této třídě zcela porozumět a refaktorovat ji. Toho jsem nemohl dosáhnout, aniž bych zvýšil její pokrytí testy. Takže jsem si napsal vlastní sadu zcela nezávislých jednotkových testů (výpis B.4, strana 380).

Když se na ně podíváte, všimnete si, že mnohé z nich jsou zakomentovány. Tyto testy neprošly. Reprezentují funkčnost, o které si myslím, že by ji třída `SerialDate` měla mít. Takže až ji refaktoruji, zapracuju i na těchto testech, aby také prošly.

I s některými zakomentovanými testy mně Clover vypisuje, že nové jednotkové testy provádějí 170 (92 procent) řádků z celkového počtu 185 proveditelných příkazů. To je docela dobré a myslím, že toto číslo ještě zvýšíme.

Několik prvních zakomentovaných rádků testu (rádky 23–63) byly tak trochu ukázkou mé domýšlivosti. Program nebyl navržen tak, aby těmito testy prošel, ale jeho funkčnost se mi zdála být zřejmá [O2]. Nejsem si jist, proč byla metoda `testWeekdayCodeToString` napsaná na prvním místě, ale protože tam je, zdá se samozřejmě, že by neměla dělat rozdíly mezi malými a velkými písmeny. Psaní těchto testů bylo triviální [T3]. Ještě jednodušší bylo je nechat projít. Stačilo změnit rádek 259 a 263 a použít `equalsIgnoreCase`.

Nechal jsem testy na řádcích 32 a 45 zakomentované, protože mi není jasné, zda podporovat zkratky, jako je „tues“ a „thurs“.

Testy na řádku 153 a 154 selhávají. Ale projít by měly [O2]. To můžeme napravit i s testy na řádcích 163 až 213 jednoduše tím, že provedeme ve funkci `stringToMonthCode` následující změny.

```
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 }
```

Zakomentovaný test na řádku 318 ukazuje chybu v metodě `getFollowingDayOfWeek` (rádek 672). Dne 25. prosince 2004 byla sobota. Následující sobota byla prvního ledna, roku 2005. Avšak když spustíme test, uvidíme, že metoda `getFollowingDayOfWeek` vrací 25. prosinec jako sobotu, která následuje po 25. prosinci. To je evidentně špatně [O3], [T1]. Problém uvidíme na řádku 685. Je to typická chyba hraniční podmínky [T5]. Měla by vypadat takto:

```
685 if (baseDOW >= targetWeekday) {
```

Je zajímavé, že tuto funkci někdo již dříve opravoval. Historie změn (rádek 43) říká, že v metodách `getPreviousDayOfWeek`, `getFollowingDayOfWeek` a `getNearestDayOfWeek` byly opraveny „mouchy“ [T6].

Jednotkový test `testGetNearestDayOfWeek` (rádek 329), který testuje metodu `getNearestDayOfWeek` (rádek 705), nebyl zpočátku tak dlouhý a vyčerpávající, jako je nyní. Přidal jsem tam mnoho testovacích případů, protože moje původní testy neprošly [T6]. Charakter selhání uvidíte, když si projdete, které testovací případy jsou zakomentovány. Tento charakter něco odhaluje [T7]. Ukazuje, že algoritmus selhává, je-li nejbližší den v budoucnosti. Je jasné, že tam je nějaký druh chyby v hraniční podmínce [T5].

Charakter testovacího prostředí z programu Clover je rovněž zajímavý [T8]. Rádek 719 se nikdy neprovádí! To znamená, že příkaz `if` z rádku 718 má vždy hodnotu false. Opravdu, když se podíváte na kód, ukáže se, že to musí být pravda. Proměnná `adjust` je vždy záporná, a tudíž nemůže být větší nebo rovna čtyřem. Takže tento algoritmus je prostě špatný.

Ten správný je zde:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
    adjust -= 7;
return SerialDate.addDays(adjust, base);
```

Nakonec lze testy na řádcích 417 a 429 obejít tak, že se prostě vyvolá výjimka `IllegalArgumentException`, místo aby se z metod `weekInMonthToString` a `relativeToString` vracej chybový řetězec.

Po těchto změnách projdou všechny jednotkové testy a nyní věřím, že třída `SerialDate` funguje. Takže nyní nastal čas ji „spravit“.

## Pak to sprav

Budeme nyní procházet třídu `SerialDate` odshora dolů a opravovat ji. Ačkoliv to nebude během diskuse vidět, po každé změně budu spouštět všechny jednotkové testy JCommon, včetně mých zdokonalených jednotkových testů. Takže v celém dalším kódu platí, že každá změna, kterou zde vidíte, funguje pro všechny testy v JCommon.

Počínaje prvním řádkem vidíme množství komentářů s licenčními informacemi, autorskými právy, autory a historii změn. Uznávám, že existují určité právní normy, o kterých je třeba se zmínit, a tak autorská práva a licence musejí zůstat. Na druhé straně je zde historie změn ještě z sedesátých let. Dnes již máme nástroje pro správu zdrojového kódu, které takovou práci dělají za nás. Historie změn by se měla smazat.

Seznam importovaných modulů na řádku 61 by se měl zkrátit a používat formu `java.text.*` a `java.util.*`. [J1]

Dívám se na formátování HTML v Javadoc (řádek 67). Mít zdrojový soubor s více jazyky mi dělá vrásky na čele. Tento komentář obsahuje čtyři jazyky. Jsou to: Java, angličtina, Javadoc a html [O1]. S takovým množstvím používaných jazyků je obtížné, aby bylo v kódu vše srozumitelné. Například výhoda pěkného umístění řádků 71 a 72 je pryč v okamžiku, kdy se pomocí Javadoc vygeneruje dokumentace. Kdo by chtěl vidět ve zdrojovém kódu značky `<ul>` nebo `<li>`? Lepší strategií by mohlo být obklopit celý komentář značkami `<pre>`, aby formátování, které je viditelné ve zdrojovém kódu, se pro Javadoc zachovalo<sup>1</sup>. Na řádku 86 je deklarace třídy. Proč se tato třída jmenuje `SerialDate`? Co je důležité ve slově „serial“? Je to proto, že je třída odvozena od `Serializable`? To se zdá pravděpodobně.

Nenechám vás hádat. Já vím proč (nebo si to alespoň myslím), proč se používá slovo „serial“. Vodítkem jsou konstanty `SERIAL_LOWER_BOUND` a `SERIAL_UPPER_BOUND` na řádcích 98 a 101. A ještě lepším vodítkem je poznámka, která začíná na řádku 830. Tato třída se jmenuje `SerialDate`, protože používá při implementaci „sériové číslo“, které znamená počet dnů od 30. prosince roku 1899. Vidím zde dva problémy. Zaprvé termín „sériové číslo“ není příliš korektní. Mohlo by to být slovíčkaření, ale znamená to spíše časový rozdíl než sériové číslo. Termín „sériové číslo“ má více

1. Ještě lepším řešením by mohlo být, kdybychom předformátovali všechny komentáře pro Javadoc, aby se jevily stejně jak v kódu, tak i dokumentu.

společného s identifikačními čísly výrobků než daty. Takže tohle jméno se mi nezdá příliš výstižné [Jm1]. Výstižnější termín by mohl být „ordinal“ (pořadový).

Druhý problém je závažnější. Název `SerialDate` te v sobě zahrnuje implementaci. Tato třída je abstraktní. Není zapotřebí, aby obsahovala cokoliv ohledně implementace. Skutečně, máme dobrý důvod implementaci skrýt! Takže zjišťuji, že tohle jméno je ve špatné úrovni abstrakce [Jm2]. Podle mého názoru by název této třídy měl být jednoduše `Date`.

Jazyk Java obsahuje bohužel příliš mnoho tříd s tímto jménem, takže to asi nebude ta nejlepší volba. Protože tato třída počítá se dny a ne s časem, uvažoval jsem o názvu `Day`, ale tohle jméno se na jiných místech používá také velmi často. Jako nejlepší kompromis jsem nakonec zvolil `DayDate`.

Odted' budu používat v této diskusi termín `DayDate`. Nechám na vás, abyste si pamatovali, že ve výpisech budete stále vidět a používat termín `Serializable`.

Chápu, proč třída `DayDate` dědí od rozhraní `Comparable` a `Serializable`. Ale proč dědí od rozhraní `MonthConstants`? Toto rozhraní (výpis B.3, strana 378) je jen hromadou statických konstant, které definují měsíce. Dědění od tříd obsahujících konstanty je starý trik programátorů v Javě, který používají proto, aby se vyhnuli výrazům, jako je `MonthConstants.January`, ale není to dobré řešení [J2]. Třída `MonthConstants` by měla ve skutečnosti být výčtovým typem.

```
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("Invalid month index " + monthIndex);  
        }  
        public final int index;  
    }  
}
```

Změna třídy `MonthConstants` na `enum` způsobí několik dalších změn ve třídě `DayDate` a všech jejich uživatelích. Tyto změny mi zabraly asi hodinu. Avšak každá funkce, která používala pro měsíc proměnnou typu `int`, používá nyní výčet `Month`. To znamená, že se můžeme zbavit metody `isValidMon-`

`thCode` (řádek 326) a všech kontrol chyb pro kódy měsíce, jako ta v metodě `monthCodeToQuarter` (řádek 356) [O5].

Dále máme na řádku 91 proměnnou `serialVersionUID`. Slouží pro ovládání serializéru. Jestliže ji změníme, jakákoliv třída `DayDate` napsaná ve starší verzi softwaru nebude čitelná a způsobí výjimku `InvalidClassException`. Ndeklarujete-li proměnnou `serialVersionUID`, překladač ji automaticky vygeneruje a bude jiná po každé změně, kterou v modulu provedete. Vím, že všechny dokumenty doporučují ruční správu této proměnné, ale mně se zdá, že automatické řízení serializace je mnohem bezpečnější [O4]. Nakonec bych raději ladil výjimku typu `InvalidClassException` než podivné chování, které by mohlo následovat, kdybych zapomněl změnit proměnnou `serialVersionUID`. Takže proměnnou vymažu – alespoň pro tentokrát<sup>2</sup>. Poznámku na řádku 93 považuju za nadbytečnou. Nadbytečné poznámky jsou pouze místa, kde se kumuluje dezinformace [K2]. Takže se jí hodláme zbavit a také všeho, co je jí podobné.

Poznámky na řádcích 97 a 100 informují o sériových číslech, o kterých jsem již mluvil dříve [K1]. Proměnné, které popisují, znamenají první a poslední validní datum, jež `DayDate` může popsat. Mohu to vysvětlit trochu lépe [Jm1].

```
public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

Není mi jasné, proč má proměnná `EARLIEST_DATE_ORDINAL` hodnotu 2 a ne 0. Na řádku 829 je náznak, který napovídá, že to má něco společného se způsobem, jakým jsou data reprezentována v MS Excel. Mnohem hlubší pohled poskytuje třída `SpreadsheetDate` (výpis B.5, strana 389), která je odvozena od `DayDate`. Poznámka na řádku 71 tuto otázku dobře popisuje.

Zdá se, že problém, který zde mám, spočívá v implementaci třídy `SpreadsheetDate` a nemá nic společného s třídou `DayDate`. Z toho přicházím k závěru, že proměnné `EARLIEST_DATE_ORDINAL` a `LATEST_DATE_ORDINAL` nepatří ve skutečnosti do třídy `DayDate` a měli bychom je přemístit do třídy `SpreadsheetDate` [O6].

Skutečně. Prohledání kódu ukazuje, že se tyto proměnné používají jen uvnitř třídy `SpreadsheetDate`. Nepoužívají se nikde ve třídě `DayDate` nebo ve kterékoliv jiné třídě šablony `JCommon`. Přesunu je tedy do třídy `SpreadsheetDate`.

Další proměnné `MINIMUM_YEAR_SUPPORTED` a `MAXIMUM_YEAR_SUPPORTED` (řádek 104 a 107) pro nás znamenají jakési dilema. Zdá se být jasné, že pokud je třída `DayDate` abstraktní a neposkytuje žádné předběžné údaje o implementaci, pak by nás neměla informovat ani o prvním nebo posledním roce. Jsem opět v pokušení přesunout tyto proměnné do třídy `SpreadsheetDate` [O6]. Avšak rychlé prohledání kódu ukazuje, že tyto proměnné používá ještě jedna třída: `RelativeDayOfWeekRule` (výpis B.6, strana 398). Vidíme to na řádkách 177 a 178 ve funkci `getDate`, kde jsou použity pro kontrolu platnosti roku u argumentu, předávaného funkci `getDate`. Dilema spočívá v tom, že uživatel abstraktní třídy potřebuje informaci ohledně její implementace.

2. Několik recenzentů tohoto textu udělalo v tomto rozhodnutí výjimku. Trvali na tom, že v případě rámce s otevřeným zdrojovým kódem bude lepší zajistit manuální kontrolu nad sériovým ID, aby menší změny v softwaru nezpůsobily, že stará serializovaná data nebudou platná. To je správný argument. Avšak selhání, jakkoliv se nám nehodí, má jasnu příčinu. Na druhé straně, pokud autor třídy zapomene aktualizovat ID, bude stav selhání nedefinovaný a může uniknout pozornosti. Myslím, že skutečně poučení z této diskuse je, že byste neměli očekávat, že se data serializovaná v jedné verzi se nebudou číst v jiné verzi.

Potřebujeme tuto informaci poskytovat, ale nechceme zaneřádit samotnou třídu DayDate. Obvykle bychom tuto implementační informaci obdrželi z instance odvozené třídy. Avšak funkci getDate se nepředává instance třídy DayDate jako parametr, ale sama takovou instanci vrací, což znamená, že ji někde musí vytvářet. Na řádcích 187 až 208 nacházíme tip. Instanci třídy DayDate vytváří jedna ze tří funkcí: getPreviousDayOfWeek, getNearestDayOfWeek nebo getFollowingDayOfWeek. Když se opět podíváme na výpis třídy DayDate, uvidíme, že všechny tyto funkce (řádky 638–724) vracejí datum vytvořené funkcí addDays (řádek 571), která volá metodu createInstance (řádek 808) a ta vytváří instanci třídy SpreadsheetDate! [O7].

Všeobecně není dobré, když základní třídy vědí o svých odvozených třídách. Abychom to opravili, měli bychom používat vzor abstraktní továrny (ABSTRACT FACTORY pattern)<sup>3</sup> a vytvořit DayDateFactory. Tato továrna bude vytvářet instance třídy DayDate, které potřebujeme pro odpovědi na otázky ohledně implementace, jako je nejnižší a nejvyšší podporované datum.

```
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);  
    }  
    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();  
    }  
}
```

Tato tovární třída nahrazuje metody createInstance jinými metodami, jako je makeDate, které mají o něco lepší jména [Jm1]. Standardní třídou je SpreadsheetDateFactory, ale lze ji kdykoliv změnit

---

3. [GOF].

a použít jinou. Statické metody, které jsou přesměrovány na abstraktní metody, používají kombinace vzoru jedináček (SINGLETON)<sup>4</sup>, dekorátor (DECORATOR)<sup>5</sup>, a abstraktní továrna (ABSTRACT FACTORY), které mohou být užitečné.

Třída `SpreadsheetDateFactory` vypadá takto:

```
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;
    }
}
```

Jak lze vidět, zde jsem již přesunul proměnnou `MINIMUM_YEAR_SUPPORTED` a `MAXIMUM_YEAR_SUPPORTED` do třídy `SpreadsheetDate`, kam patří [O6]. Dalším krokem ve třídě `DayDate` jsou konstanty, které začínají na řádku 109. To by měly být opět hodnoty výčtového typu typu [J3]. Tento vzor jsme již viděli dříve, takže se zde nebudu opakovat. Uvidíte je ve finálních výpisech.

Dále zde vidíme několik polí, začínajících na řádku 140 polem `LAST_DAY_OF_MONTH`. První mou stárostí s těmito poli bude, že jejich popisné komentáře jsou nadbytečné [K3]. Jejich jména jsou dostatečně popisná. Takže tyto komentáře smažu.

Zdá se, že neexistuje důvod, proč by tato pole nemohla být soukromá [O8], když je tam statická funkce `lastDayOfMonth`, která poskytuje tytéž údaje.

Další pole `AGGREGATE_DAYS_TO_END_OF_MONTH` je poněkud tajemnější, protože se nikde v rámci `JCommon` nepoužívá [O9]. Takže ho smažu.

Totéž platí pro `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

---

4. [GOF].

5. [GOF].

Další pole AGGREGATE\_DAYS\_TO\_END\_OF\_PRECEDING\_MONTH se používá pouze ve třídě SpreadsheetDate (řádky 434 a 473). To vyvolává otázku, zda bychom ho neměli do této třídy přesunout. Argument proč ne spočívá v tom, že tohle pole není specifické pro žádnou konkrétní implementaci [O6]. Na druhé straně neexistuje žádná jiná implementace kromě SpreadsheetDate, takže bychom měli pole přesunout co nejblíže místu, kde se používá [O10].

Tento spor vyřeší požadavek konzistence [O11], podle kterého bychom měli vytvořit pole jako soukromé a dát ho k dispozici pomocí funkce, jako je julianDateOfLastDayOfMonth. Nezdá se, že by taková funkce byla zapotřebí někde jinde. Navíc pole můžeme přesunout jednoduše zpátky do třídy DayDate, pokud by to nějaká nová implementace DayDate potřebovala. Přesunul jsem je.

Totéž platí pro pole LEAP\_YEAR\_AGGREGATE\_DAYS\_TO\_END\_OF\_MONTH.

Dále zde vidíme tři sady konstant, z nichž bychom mohli udělat výčtový typ (řádky 162–205). První z nich vybírá týden v měsíci. Změnil jsem ji na výčtový typ jménem WeekInMonth.

```
public enum WeekInMonth {
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
    public final int index;
    WeekInMonth(int index) {
        this.index = index;
    }
}
```

Druhá sada konstant (řádky 177–187) je poněkud obskurnější. Konstanty INCLUDE\_NONE, INCLUDE\_FIRST, INCLUDE\_SECOND, a INCLUDE\_BOTH určují, zda koncové datum nějakého intervalu rozsahu by mělo být v tomto intervalu zahrnuto nebo ne. Matematicky mluvíme o termínu „otevřený interval“, „polootevřený interval“ a „uzavřený interval“. Myslím, že je mnohem srozumitelnější používat matematické názvosloví [Jm3] a tak jsem změnil typ na výčtový typ se jménem DateInterval a s prvky CLOSED, CLOSED\_LEFT, CLOSED\_RIGHT, a OPEN.

Třetí sada konstant (řádky 189–205) popisuje, zda by mělo hledání určitého dne v týdnu vést na minulý, příští nebo nejbližší instanci. Jak to pojmenovat, je přinejlepším obtížné. Nakonec jsem se rozhodl pro název WeekdayRange s výčtovými položkami LAST, NEXT, a NEAREST.

Jména, která jsem vybral, se vám nemusí líbit. Mně smysl dávají, ale nemusí dávat smysl vám. Hlavní je, že nyní mají tvar, který můžete snadno změnit [J3]. Již se nepředávají jako celá čísla, ale jako symboly. Mohu použít funkci z mého vývojového prostředí „změň jméno“, která mění jména nebo typy, aniž bychom se museli zabývat tím, že jsme někde v kódu zapomněli -1 či 2, nebo že nějaká deklarace celočíselného argumentu zůstala špatně popsaná.

Popisné pole na řádku 208 se zřejmě nikde nepoužívá. Smazal jsem ho i s jeho přístupovými a modifikujícími metodami [O9].

Také jsem smazal zbytečný implicitní konstruktor na řádku 213 [O12]. Vygeneruje ho za nás překladač.

Metodu isValiWeekdayCode (řádky 216–238) můžeme přeskočit, protože jsme ji smazali při vytváření výčtu Day.

To nás přivádí k metodě stringToWeekdayCode (řádky 242–270). Řádky Javadoc nepřispívají k signatuře metody ničím pozitivním a zde jen překážeji [K3], [O12]. Jedinou užitečnou věcí, kterou sem

vnáší Javadoc, je popis vracené hodnoty -1. Protože jsme ale předělali `Day` na výčtový typ, je komentář vlastně chybný. Nyní tato metoda vyvolává výjimku typu `IllegalArgumentException`. Takže jsem vymazal komentáře Javadoc.

Dále jsem smazal ve všech deklaracích argumentů a proměnných klíčová slova `final`. Pokud vím, neznamenají žádnou hodnotu, ale podílejí se na zaneřádění kódu [O12]. Odstranění klíčových slov `final` je v určitém rozporu s konvenčním uvažováním. Například Robert Simmons<sup>6</sup> nám důrazně doporučuje „...používejte slovo `final` všude ve svém kódu“. Naprosto nesouhlasím. Myslím si, že tohle slovo má svůj význam, jako třeba při deklaraci příležitostních konstant ale jinak má malý význam a podílí se na zaneřádění kódu. Možná, že takto uvažuji proto, že druh chyb, které může slovo `final` zachytit, je již zachycen v jednotkových testech, které příšu.

Nepotřeboval jsem dva příkazy `if` [O5] uvnitř cyklu `for` (rádek 259 a 263) a tak jsem je spojil do jednoho příkazu `if` a použil jsem operátor `||`. Také jsem použil výčet `Day` pro řízení cyklu `for` a provedl jsem několik kosmetických změn. Připadalo mi, že tato metoda do třídy `DayDate` vlastně nepatří. Je to ve skutečnosti funkce pro syntaktickou analýzu výčtu `Day`. Přesunul jsem ji tedy do výčtu `Day`. Avšak tím nám objem tohoto výčtu pěkně narostl. Protože jeho koncepce nezávisí na třídě `DayDate`, přesunul jsem `Day` mimo třídu `DayDate` do samostatného zdrojového souboru [O13].

Do výčtu `Day` jsem také přesunul funkci `weekdayCodeToString` (rádky 272–286) a nazval jsem ji `toString`.

```
public enum Day {
    MONDAY(Calendar.MONDAY),
    TUESDAY(Calendar.TUESDAY),
    WEDNESDAY(Calendar.WEDNESDAY),
    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("Illegal day index: %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()) {
```

6. [Simmons04], str. 73.

```

        if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
            s.equalsIgnoreCase(weekDayNames[day.index])) {
            return day;
        }
    }
    throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
}
public String toString() {
    return dateSymbols.getWeekdays()[index];
}
}

```

Máme dvě funkce `getMonths` (řádky 288–316). První z nich volá druhou, kterou pak už nevolá žádná jiná. Sloučil jsem je tedy do jedné a kód jsem značně zjednodušil [O9],[O12],[F4]. Nakonec jsem jí změnil jméno, aby bylo popisnější [Jm1].

```

public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
}

```

Funkce `isValidMonthCode` (řádky 326–346) je nyní zbytečná, a to díky výčtu `Month`. Smazal jsem ji [O9].

U funkce `monthCodeToQuarter` (řádky 356–375) cítíme hlubší problém tzv. chybějících schopností (FEATURE ENVY)<sup>7</sup>. [O14] a pravděpodobně patří do výčtu `Month` jako metoda `quarter`. Vyměnil jsem ji.

```

public int quarter() {
    return 1 + (index-1)/3;
}

```

Tím se výčet `Month` natolik rozrostl, že může být samostatnou třídou. Vyjmul jsem jej tedy z třídy `DayDate`, aby zůstala konzistentní s výčtem `Day` [O11], [O13].

Další dvě metody se jmenují `monthCodeToString` (řádky 377–426). Opět zde vidíme vzor, kdy jedna metoda volá své dvojče s příznakem. Předávat logickou hodnotu jako argument funkce není obecně dobrý způsob, zvláště když pouze určuje formát výstupní hodnoty [O15]. Tyto funkce jsem přejmenoval, zjednodušil, restrukturalizoval a poté je přesunul do výčtu `Month` [Jm1],[Jm3],[K3],[O14].

```

public String toString() {
    return dateFormatSymbols.getMonths()[index - 1];
}
public String toShortString() {
    return dateFormatSymbols.getShortMonths()[index - 1];
}

```

Následující metoda je `stringToMonthCode` (řádky 428–472). Přejmenoval jsem ji, přesunul do výčtu `Month` a zjednodušil [Jm1], [Jm3], [K3], [O14], [O12].

---

7. [Refactoring].

```

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("Invalid month " + s);
}
private boolean matches(String s) {
    return s.equalsIgnoreCase(toString()) ||
           s.equalsIgnoreCase(toShortString());
}

```

Metoda `isLeapYear` (řádky 495–517) by mohla být trochu expresivnější [O16].

```

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);
}

```

Následující funkce `leapYearCount` (řádky 519–536) nepatří ve skutečnosti do třídy `DayDate`. S výjimkou dvou metod ve třídě `SpreadsheetDate` ji nic nevolá. Přemístil jsem ji tedy dolů [O6].

Poslední funkce `lastDayOfMonth` (řádky 538–560) používá pole `LAST_DAY_OF_MONTH`. To ve skutečnosti patří do výčtu `Month` [O17], a tak jsem jej tam přemístil. Také jsem tuto funkci zjednodušil a zexpresivnil [O16].

```

public static int lastDayOfMonth(Month month, int year) {
    if (month == Month.FEBRUARY && isLeapYear(year))
        return month.lastDay() + 1;
    else
        return month.lastDay();
}

```

Nyní to začíná být zajímavé. Další funkci je `addDays` (řádky 562–576). Za prvé platí, že pracuje-li tato funkce s proměnnými třídy `DayDate`, neměla by být statická [O18]. Udělal jsem z ní tedy instanční metodu. Za druhé, volá funkci `toSerial`. Tato funkce by se měla přejmenovat na `toOrdinal` [Jm1]. A nakonec lze tuto metodu zjednodušit.

```

public DayDate addDays(int days) {
    return DayDateFactory.makeDate(toOrdinal() + days);
}

```

Totéž platí pro funkci `addMonths` (řádky 578–602). Měla by to být instanční metoda [O18]. Její algoritmus je poněkud komplikovaný, a tak jsem použil metodu vysvětlující dočasné proměnné (EXPLAINING TEMPORARY VARIABLES – zavedení nových dočasných proměnných za účelem snadněj-

šího pochopení složitějších výrazů. Pozn. překl.)<sup>8</sup>, aby byl srozumitelnější. Přejmenoval jsem také metodu `getYYY` na `getYear` [Jm1].

```
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 lastDayOfMonth = lastDayOfMonth(resultMonth, resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfMonth);
    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

Funkce `addYears` (řádky 604–626) nepřináší ve srovnání s ostatními funkcemi nic nového.

```
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);
}
```

Mám trochu svrbění v hloubi duše, zda mám změnit tyto metody ze statických na instanční. Umí výraz `date.addDays(5)` vysvětlit skutečnost, že se objekt `date` nemění a že návratovou hodnotou je nová instance třídy `DayDate`? Nebo z toho vyplývá, že k objektu `date` přičítáme pět dnů, což je chyba? Možná si myslíte, že to není žádný velký problém, ale část kódu, která následuje, může být velmi zavádějící [O20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // bump date by one week.
```

Čtenář tohoto kódu by mohl vzít jako fakt, že funkce `addDays` mění objekt `date`. Potřebujeme tedy název, který tuto dvojznačnost odstraní [Jm4]. Změnil jsem názvy na `plusDays` a `plusMonths`. Zdá se mi, že záměr této metody je dobře podchycen v následujícím řádku:

```
DayDate date = oldDate.plusDays(5);
```

zatímco z následujícího textu nevyplývá dostatečně souvislost, ze které by měl čtenář jednoduše pochopit, že se mění objekt `date`:

```
date.plusDays(5);
```

Algoritmus je čím dál tím zajímavější. Funkce `getPreviousDayOfWeek` (řádky 628–660) funguje, ale je trochu komplikovanější. Po určitých úvahách o tom, co má vlastně dělat [O21], jsem byl schopen ji zjednodušit a použít metodu dočasných vysvětlujících proměnných [O19], aby byla srozumitelnější. Také jsem ji změnil ze statické metody na instanční [O18] a zbavil se všech zdvojených instančních metod [O5] (řádky 997–1008).

---

8. [Beck97].

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget >= 0)
        offsetToTarget -= 7;
    return plusDays(offsetToTarget);
}
```

Stejně výsledky bude mít analýza funkce `getFollowingDayOfWeek` (řádky 662–693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget <= 0)
        offsetToTarget += 7;
    return plusDays(offsetToTarget);
}
```

Další funkcí je `getNearestDayOfWeek` (řádky 695–726), kterou jsme opravovali na straně 276 již dříve. Ale změny, které jsem tam provedl, nejsou v případě dvou posledních funkcí konzistentní s aktuálním modelem [O11]. Opravil jsem je a použil metodu dočasných vysvětlujících proměnných (EXPLAINING TEMPORARY VARIABLES) [O19], aby byl algoritmus srozumitelnější.

```
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` (řádky 728–740) je poněkud zvláštní, protože je to instanční metoda, u které dochází k tzv. nedostatku schopností (FEATURE ENVY) vlastní třídy tím, že přebírá třídu `DayDate` jako argument. Udělal jsem z ní skutečnou instanční metodu a některá jména jsem změnil na srozumitelnější.

```
public DayDate getEndOfMonth() {
    Month month = getMonth();
    int year = getYear();
    int lastDay = lastDayOfMonth(month, year);
    return DayDateFactory.makeDate(lastDay, month, year);
}
```

Refaktorování metody `weekInMonthToString` (řádky 742–761) bylo velmi zajímavé. Použil jsem nástroj pro refaktorování z mého integrovaného vývojového prostředí a jako první krok jsem přesunul tuto metodu do výčtu `WeekInMonth`, který jsem vytvořil na straně 281. Poté jsem ji přejmenoval na `toString`. V dalším kroku jsem ji změnil ze statické metody na instanční. Všechny testy stále procházely. (Dokážete uhodnout, kam mířím?)

V dalším kroku jsem celou metodu vymazal! Pět asercí selhalo (řádky 411–415, výpis B.4, strana 380). Tyto řádky jsem změnil a použil názvů výčtových prvků (`FIRST`, `SECOND`, ...). Všechny testy prošly. Víte proč? Víte také, proč byl každý z těchto kroků nezbytný? Nástroj pro refaktorování

zajistil, že všechny dřívější objekty, které volaly funkci `weekInMonthToString`, volaly nyní metodu `toString` z výčtu `weekInMonth`, protože všechny výčty implementují funkci `toString`, aby vracely svá jména...

Bohužel jsem byl příliš chytrý. Ať byl tento sled refaktorování jakkoli báječný, nakonec jsem zjistil, že jedinými uživateli této funkce byly testy, které jsem právě modifikoval, a tak jsem je smazal.

Jen hlupák udělá stejnou chybu dvakrát. Po zjištění, že se funkce `relativeToString` (řádky 765–781) volá jen z testů, jsem funkci i s testy jednoduše smazal.

Nakonec jsme se dostali k abstraktním metodám této abstraktní třídy. A hned první se nám hodí: `toSerial` (řádky 838–844). Na straně 284 jsem změnil její jméno na `toOrdinal`. Když jsem se na ni podíval v tomto kontextu, rozhodl jsem se její jméno změnit na `getOrdinalDay`.

Další abstraktní metoda je `toDate` (řádky 838–844). Provádí konverzi třídy `DayDate` na `java.util.Date`. Proč je tato metoda abstraktní? Podíváme-li se na její implementaci ve třídě `SpreadsheetDate` (řádky 198–207, výpis B.5, strana 389), uvidíme, že nezávisí na žádné části implementace této třídy [O6]. Tak jsem ji přemístil výše.

Metody `getYYYY`, `getMonth` a `getDayOfMonth` jsou abstraktní. Avšak metoda `getDayOfWeek` je další z těch, které by se měly ze třídy `SpreadSheetDate` přemístit směrem nahoru, protože nezávisí na ničem, co bychom našli v `DayDate` [O6]. Nebo snad ano?

Když se na ně podíváte pečlivě (řádek 247, výpis B.5, strana 389), uvidíte, že algoritmus závisí implicitně na tom, kde pořadová čísla dnů začínají (jinými slovy, jaký den v týdnu má číslo 0). Takže i když tato funkce nemá žádné fyzické závislosti, které by bylo možné přemístit do třídy `DayDate`, obsahuje závislosti logické.

Logické závislosti tohoto typu mě trápí [O22]. Pokud něco logického závisí na implementaci, pak by na ní mělo záviset i něco fyzického. Také se mi zdá, že by samotný algoritmus mohl být genericky s mnohem menší částí, která by byla závislá na implementaci [O6].

Vytvořil jsem tedy abstraktní metodu ve třídě `DayDate` jménem `getDayOfWeekForOrdinalZero` a implementoval ji ve třídě `SpreadsheetDate`, aby vracela hodnotu `Day.SATURDAY`. Pak jsem přesunul metodu `getDayOfWeek` nahoru do třídy `DayDate` a změnil ji tak, aby volala metody `getOrdinalDay` a `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {  
    Day startingDay = getDayOfWeekForOrdinalZero();  
    int startingOffset = startingDay.index - Day.SUNDAY.index;  
    return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);  
}
```

Mimochodem, podívejte se pečlivě na poznámku z řádků 895 až 899. Bylo tohle opakování opravdu nutné? Jako obvykle jsem tuto poznámku spolu s dalšími vymazal.

Dalším krokem byla metoda `compare` (řádky 902–913). U této metody je opět nevhodné, že je abstraktní [O6], a tak jsem její implementaci posunul vzhůru do třídy `DayDate`. Ani její jméno toho mnoho neříká [Jm1]. Tato metoda v podstatě vrací rozdíl ve dnech, které uplynuly od hodnoty dané argumentem. Změnil jsem tedy její jméno na `daysSince`. Také jsem si všiml, že jsme pro tuto metodu neměli žádné testy, a tak jsem je napsal.

Následujících šest funkcí (řádky 915–980) byly všechno abstraktní metody, které by měly být implementovány v DayDate. Přesunul jsem tedy všechny vzhůru z třídy SpreadsheetDate.

Poslední funkci `isInRange` (řádky 982–995) je rovněž potřeba přesunout směrem vzhůru a refaktorovat. Příkaz `switch` je poněkud nepřijemný [O23] a lze jej nahradit tím, že jednotlivé případy převedeme do výčtu `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);
}
```

To nás přivádí ke konci třídy `DayDate`. Ještě jednou si tedy celou třídu projdeme a podíváme se na to, jak funguje.

Za prvé je úvodní poznámka již dlouho zastaralá, a tak jsem ji zkrátil a opravil [K2].

V dalším kroku jsem přemístil veškeré zbývající výčty do samostatných souborů [O12].

V dalším kroku jsem přesunul statickou proměnnou (`dateFormatSymbols`) a tři statické metody (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) do nové třídy jménem `DateUtil` [O6].

Abstraktní metody patří na začátek, a tak jsem je přesunul směrem nahoru [O24].

Změnil jsem výčet `Month.make` na `Month.fromInt` [Jm1] a totéž jsem provedl se všemi ostatními výčty.

Pro všechny výčty jsem vytvořil přístupovou metodu `toInt()` a předělal datovou složku `index` na soukromou.

V metodách `plusYears` a `plusMonths` bylo jedno zajímavé zdvojení [O5], které jsem dokázal extrahovat do nové metody jménem `correctLastDayOfMonth`. Tím se staly všechny tři metody čitelnější.

Zbavil jsem se magického čísla 1 [O25]. Podle potřeby jsem je nahradil metodou `Month.JANUARY.toInt()` nebo `Day.SUNDAY.toInt()`. Věnoval jsem nějaký čas třídě `SpreadsheetDate` a trochu její algoritmus vyčistil. Výsledek je ve výpisech B.7, strana 402 až B.16, strana 411.

Je zajímavé, že pokrytí kódu ve třídě `DayDate` se snížilo na 84,9 procenta! Není to proto, že bychom testovali méně funkcionality, ale spíše proto, že se třída natolik zkrátila, že několik nepokrytých řádků nyní tvoří větší procento. Nyní je 45 z 53 proveditelných řádků ve třídě `DayDate` pokryto testy. Ty ostatní jsou natolik triviální, že nestojí za to je testovat.

## Závěr

Tak jsme se opět jednou řídili skautským pravidlem. Zanechali jsme kód o trochu čistější, než byl na počátku. Věnovali jsme tomu nějaký čas, ale vyplatilo se to. Zvýšilo se pokrytí kódu testy, byly odstraněny některé chyby, kód se zkrátil a je srozumitelnější. Další člověk, který bude s kódem pracovat, to bude mít určitě snadnější, než jsme to měli my. Je pravděpodobné, že jej bude umět vyčíslet o něco lépe než my.

## Použitá literatura

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

[Simmons04]: *Hardcore Java*, Robert Simmons, Jr., O'Reilly, 2004.

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

[Beck97]: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.



# KAPITOLA 17

## Skryté problémy a heuristika

**V této kapitole najdete:**

- ◆ Komentáře
- ◆ Prostředí
- ◆ Funkce
- ◆ Obecné
- ◆ Java
- ◆ Jména
- ◆ Testy
- ◆ Závěr
- ◆ Použitá literatura



Martin Fowler identifikoval ve své vynikající knize *Refaktorování*<sup>1</sup> mnoho různých „skrytých problémů kódu“. Seznam, který následuje, obsahuje mnoho Martinových postřehů a také mnoho mých. Obsahuje i jiné perly a heuristiky, které ve své profesi používám.

Tento seznam jsem vytvořil tak, že jsem prošel několik různých programů a refaktoroval je. Po každé změně jsem se ptal sám sebe, *proč jsem tyto změny provedl, a důvod jsem si zapsal*. Výsledkem je celkem obsáhlý seznam potenciálních problémů, jejichž existenci pociťuji, kdykoliv nějaký kód procházím.

Tento seznam můžete číst systémem odshora dolů, můžete ho ale použít i jako referenční příručku.

## Komentáře

### K1: Nevhodné komentáře

Není vhodné mít v komentáři informace, které by měly být jinde, například v systému pro správu zdrojového kódu, v systému pro sledování problémů nebo v jakémkoliv jiném podobném záznamovém systému. Například historie změn dokáže zaneřádit zdrojové soubory množstvím historického a nezajímavého textu. Obecně by se v komentářích neměla objevovat metadata, jako jsou jména autorů, datum poslední modifikace, číslo problému atd. Ty by měly zůstat vyhrazeny pro technické poznámky týkající se kódu a jeho návrhu.

### K2: Zastaralé komentáře

Komentář, který je starého data, který je irrelevantní, nebo chybný, je zastaralý. Komentáře zastarávají velmi rychle. Komentáře, které v budoucnu zastarají, je nejlepší vůbec nepsat. Pokud na takový komentář narazíte, bud' jej aktualizujte, nebo se jej co nejrychleji zbabte. Zastaralé komentáře většinou odbíhají od tématu a významu kódu, který kdysi popisovaly. Stávají se plovoucími ostrovky bezvýznamnosti a špatnými ukazateli při procházení kódem.

### K3: Nadbytečný komentář

Komentář je nadbytečný, když popisuje něco, co se dostatečně popisuje samo. Například:

```
i++; // increment i
```

Jiným příkladem jsou dokumentační komentáře, které neříkají o nic víc (nebo dokonce ještě méně), než co obsahuje signatura funkce:

```
/**  
 * @param sellRequest  
 * @return  
 * @throws ManagedComponentException  
 */  
public SellResponse beginSellItem(SellRequest sellRequest)  
throws ManagedComponentException
```

Komentáře by měly sdělovat věci, které samotný kód vyjádřit nemůže.

---

1. [Refactoring].

## K4: Špatně napsaný komentář

Má-li smysl psát nějaký komentář, pak stojí za to napsat ho dobré. Jestliže chcete psát nějaký komentář, věnujte čas tomu, aby to byl ten nejlepší komentář, jaký umíte. Volte pečlivě slova. Používejte správně gramatiku i interpunkci. Mluvte souvisle. Nemluvte o samozřejmém. Buďte struční.

## K5: Zakomentovaný kód

Když vidíme příliš mnoho zakomentovaného kódu, dohání mě to k šílenství. Kdo ví, jak je tento kód starý? Kdo ví, zda je k něčemu dobrý? Ale nikdo jej nevymaže, protože každý předpokládá, že jej bude potřebovat někdo jiný.

Takový kód tam zůstává, degeneruje a s každým následujícím dnem se stává méně a méně relevantním. Volá funkce, které již neexistují. Používá proměnné, jejichž jména se dávno změnila. Řídí se konvencemi, které jsou zastaralé. Zaplňuje moduly, ve kterých trčí a odrazuje každého, kdo by jej chtěl číst. Zakomentovaný kód je *ohavnost*.

Pokud uvidíte zakomentovaný kód, *vymaže ho!* Nelamte si s tím hlavu. Systém pro správu zdrojového kódu si ho pamatuje. Bude-li jej kdokoliv potřebovat, může se vrátit k předchozí verzi. Nedovolte, aby zakomentovaný kód dlouho přežíval.

## Prostředí

### P1: Sestavení vyžaduje více než jeden krok

Sestavení projektu by mělo být jednoduché a triviální. Není dobré, musíte-li zaškrťat spoustu krátkých částí ze systému pro správu zdrojového kódu. Neměli byste potřebovat sekvence takuplných příkazů nebo kontextově závislých skriptů, abyste mohli sestavit jednotlivé elementy. Neměli byste být nuceni dohledávat tady a támhle všelijaké další soubory JAR, XML a jiné artefakty, které systém vyžaduje. Měli byste být schopni vyjmout program ze systému jediným klepnutím a pomocí dalšího jednoduchého příkazu provést jeho sestavení.

```
svn get mySystem  
cd mySystem  
ant all
```

### P2: Testy vyžadují více než jeden krok

Měli byste být schopni spustit *všechny* jednotkové testy jediným příkazem. V nejlepším případě můžete tyto testy ve svém IDE spustit klepnutím na jedno tlačítko. Při nejhorším byste je ve svém uživatelském prostředí měli spustit pomocí jednoho nekomplikovaného příkazu. Schopnost spustit všechny testy je natolik zásadní a důležitá, že by to mělo být rychlé, jednoduché a samozřejmé.

## Funkce

### F1: Příliš mnoho argumentů

Funkce by měly mít malý počet argumentů. Nejlépe je nemít žádný, pak následuje jeden, dva nebo tři argumenty. Mít více než tři argumenty je velmi problematické a měli bychom se tomu bez váhání vyhýbat. (Viz „Argumenty funkci“ na straně 61.)

## F2: Výstupní argumenty

Výstupní argumenty neodpovídají lidské intuici. Čtenáři očekávají, že argumenty budou vstupní, ne výstupní. Pokud musí funkce změnit nějaký stav, ať méně stav objektu, který ji volá. (Viz „Výstupní argumenty“ na straně 65.)

## F3: Logické argumenty

Logické argumenty hlasitě signalizují, že funkce provádí více než jen jednu činnost. Jsou matoucí a neměly by se používat. (Viz „Logické argumenty“ na straně 62.)

## F4: Mrtvé funkce

Metody, které nikdo nevolá, by se měly zrušit. Udržovat nepoužívaný kód je nákladné. Nebojte se takové funkce smazat. Mějte na paměti, že váš systém pro správu zdrojového kódu je bude mít někde uložené.

# Obecné

## O1: Více jazyků v jednom zdrojovém souboru

Dnešní moderní programová prostředí umožňují umístit do jednoho zdrojového souboru úseky s více jazyky. Například javový zdrojový soubor může obsahovat kód v XML, HTML, YAML, dokumentaci pro JavaDoc, angličtinu, zdrojový kód v JavaScriptu atd. Nebo uvedu jiný příklad. Soubor HTML a JSP může obsahovat Javu, syntaxi knihovních značek atd. To je přinejlepší matoucí a přinejhorším lehkomyslné a nedbalé. Ideální je, když bude zdrojový kód obsahovat jeden a pouze jeden jazyk. Budeme-li realističtí, budeme pravděpodobně muset používat více jazyků. Ale měli bychom ve zdrojových souborech usilovat o minimalizaci jak počtu ostatních jazyků, tak i jejich rozsahu.

## O2: Není implementováno to, co je samozřejmé

Podle „Principu nejmenšího překvapení“<sup>2</sup> by měla jakákoli funkce nebo třída implementovat takovou funkčnost, kterou by jiný programátor mohl logicky očekávat. Uvažujme například o funkci, která převádí názvy dnů na prvky výčtu, reprezentujícího dny.

```
Day day = DayDate.StringToDate(String dayName);
```

Předpokládali bychom, že řetězec „Monday“ bude převeden na Day.MONDAY. Také bychom očekávali, že bude akceptovat již používané zkratky dnů a že nebude dělat rozdíly mezi malými a velkými písmeny.

Pokud není implementována snadno předvídatelná funkčnost, čtenáři a uživatelé kódu se nemohou spoléhat na intuitivní chápání názvů funkcí. Přestanou se spoléhat na původního autora a musí se vrátit zpět ke čtení detailů kódu.

2. „The Principle of Least Surprise,” nebo “The Principle of Least Astonishment”: [http://en.wikipedia.org/wiki/Principle\\_of\\_least\\_astonishment](http://en.wikipedia.org/wiki/Principle_of_least_astonishment).

## O3: Nekorektní funkčnost na hranicích

Tvrzení, že se má kód chovat korektně, vypadá jako samozřejmost. Problémem je, že si jen zřídka uvědomujeme, jak je korektní chování komplikované. Vývojáři píšou často funkce, o kterých si myslí, že budou fungovat, a pak raději věří své intuici, než by vynaložili úsilí a ověřili si, že jejich kód funguje i v mimořádných případech a za mezních okolností.

Zde nelze náležitou píliničím nahradit. Každá mezní podmínka, každý mimořádný případ, každé zvláštní chování nebo výjimka představují nebezpečí, kterým může být jakkoliv elegantní a intuitivní algoritmus znehodnocen. *Nespolehejte se na svou intuici.* Prověřte si každou mezní podmínu a napište pro ni testy.

## O4: Zrušená zabezpečení

Reaktor v Černobylu se roztavil, protože ředitel závodu postupně zrušil veškeré bezpečnostní mechanismy. Tato zabezpečení komplikovala spuštění experimentu. Výsledkem bylo, že místo experimentu uviděl svět první vekou civilní nukleární katastrofu.

Rušit bezpečnostní opatření je riskantní. Použití ručního ovládání `serialVersionUID` může být nezbytné, ale vždy bude znamenat riziko. Vypnutí některých varování překladače (nebo dokonce všech!) může být užitečné k tomu, aby proběhlo sestavení, ale za cenu nekonečného počtu ladících cyklů. Vypnout selhávající testy a namlouvat si, že projdou později, je totéž, jako předstírat, že vaše kreditní karta jsou hotové peníze.

## O5: Zdvojení

Zásada o škodlivosti zdvojeného kódu je v této knize jednou z nejdůležitějších a měli byste ji brát velmi vážně. Prakticky každý autor, který se zabývá projektováním softwaru, se o ní zmíňuje. Dave Thomas a Andy Hunt ji nazvali „neopakujte se“ (DRY – Don't repeat yourself)<sup>3</sup>. Kent Beck z ní udělal jedním ze stěžejních principů extrémního programování a nazval ji „Jednou a jen jednou.“ Ron Jeffries staví tohle pravidlo na druhé místo hned za požadavek, aby všechny testy procházely.

Každé zdvojení kódu znamená, že jste možná přehlédl možnost, aby vznikla nějaká abstrakce. Zdvojení se pravděpodobně může změnit v subrutinu nebo přímo v nějakou třídu. Když zdvojení převedete na abstrakci, rozšíříte slovník jazyka svého návrhu. Abstraktní prostředky, které jste takto vytvořili, pak mohou používat jiní programátoři. Kódování bude rychlejší a méně náchylné k chybám, protože jste zvýšili úroveň abstrakce.

Nejviditelnější formou zdvojení kódu jsou úseky identického kódu, které vypadají, jako by se nějaký programátor zbláznil a vkládal neustále týž kód. Ten je třeba nahradit jednoduchými metodami.

Méně viditelnou formou jsou série příkazů `switch/case` nebo `if/else`, které se opakovaně objevují v různých modulech a vždy testují stejnou množinu podmínek. Ty je třeba nahradit polymorfismem.

Ještě méně nápadné jsou moduly, které mají podobné algoritmy, ale jejich kód je odlišný. To je stále zdvojení a mělo by se řešit použitím šablonové metody (TEMPLATE METHOD)<sup>4</sup> nebo návrhového vzoru strategie (STRATEGY)<sup>5</sup>. Ve skutečnosti představuje většina návrhových vzorů z posledních pat-

3. [PRAG].
4. [GOF].
5. [GOF].

nácti let dobře známé metody určené k eliminaci zdvojeného kódu. Také Coddovy normální formy jsou strategiemi, jak eliminovat zdvojení v databázových schématech. Objektově orientované programování je strategie organizace modulů a eliminace zdvojení. Není divu, že je tomu tak i u strukturovaného programování. Myslím, že to hlavní bylo řešeno. Vyhledávejte a odstraňujte zdvojený kód všude, kde je to možné.

## 06: Kód na špatné úrovni abstrakce

Je důležité vytvářet abstrakce, které oddělují pojmy vyšší úrovně od detailů nižší úrovně. Někdy vytváříme abstraktní třídy, které mají obsahovat pojmy vyšší úrovně, a třídy odvozené, jež mají obsahovat pojmy nižší úrovně. Poté se potřebujeme přesvědčit, že je tato separace úplná. Chceme, aby *všechny* pojmy nižší úrovně byly umístěny v odvozených třídách a aby *všechny* pojmy vyšší úrovně byly v základních třídách.

Například konstanty, proměnné nebo pomocné funkce, které patří k detailní implementaci, by neměly být v základní třídě. Základní třída by o nich neměla mít žádné informace.

Toto pravidlo platí i pro zdrojové soubory, komponenty a moduly. Dobrý návrh softwaru vyžaduje, abychom separovali pojmy na různých úrovních a umisťovali je do různých kontejnerů. Někdy tyto kontejnery tvoří základní třídy nebo třídy odvozené a někdy zdrojové soubory, moduly nebo komponenty. Ať už je to jakkoliv, separace musí být úplná. Nechceme, aby byly pojmy nízké a vysoké úrovně smíchány dohromady.

Podívejte se na následující kód:

```
public interface Stack {  
    Object pop() throws EmptyException;  
    void push(Object o) throws FullException;  
    double percentFull();  
    class EmptyException extends Exception {}  
    class FullException extends Exception {}  
}
```

Funkce `percentFull` je na špatné úrovni abstrakce. Přestože existuje mnoho implementací rozhraní `Stack`, u kterých je pojem *fullness* (zaplněnost) opodstatněný, existují jiné implementace, jež prostě nemohly mít žádnou informaci o tom, nakolik jsou zaplněny. Proto by bylo lepší umístit funkci do odvozeného rozhraní, jako je např. `BoundedStack`.

Možná se vám zdá, že v případě neomezeného zásobníku by implementace prostě mohla vracet nulu. Problém ale je, že ve skutečnosti žádný zásobník neomezený není. Nemůžete reálně zabránit vzniku výjimky `OutOfMemoryException` tím, že budete hlídat hodnotu funkce `stack.percentFull() < 50.0`.

Implementovat funkci, která vrací nulu, znamená poskytovat nepravdivé informace.

Smyslem je, abyste neposkytovali nepravdivé informace nebo hledali falešná východiska ze špatně umístěné abstrakce. Izolování abstrakcí je jednou z nejobtížnějších záležitostí, které vývojáři řeší, a když se dopustíte chyby, oprava není krátkodobou záležitostí.

## O7: Základní třídy, které závisejí na odvozených třídách

Nejobecnějším důvodem pro rozdělení pojmu mezi základní a odvozené třídy je, aby pojmy vyšší úrovně v základní třídě nezávisely na pojmech nižší úrovně ve třídách odvozených. Z toho plyně, že když základní třídy obsahují názvy svých odvozených tříd, můžeme se domnívat, že tam něco nebude v pořádku. Obecně platí, že základní třídy by neměly vědět nic o svých odvozených třídách.

Toto pravidlo má samozřejmě výjimky. Někdy je počet odvozených tříd striktně omezen a základní třídy obsahují kód, který jednotlivé podtřídy vybírá. Tohle lze často vidět u implementací konečných stavových automatů. Avšak v takovém případě jsou odvozené třídy se základní třídou silně svázány a vždy jsou nasazovány společně ve stejném souboru jar. V obecném případě chceme, aby bylo možné odvozené a základní třídy nasazovat v různých souborech jar.

Nasazování odvozených a základních tříd v různých souborech jar s tím, že neobsahují žádné informace o odvozených souborech jar, nám umožňuje nasazovat své systémy v samostatných a nezávislých komponentách. Když jsou takové komponenty modifikovány, mohou být nasazovány opakováně, aniž by bylo nutné nasazovat základní komponenty. To znamená, že dopad změn je mnohem nižší a údržba systému v terénu je jednodušší.

## O8: Příliš mnoho informací

Dobře definované moduly mají velmi malá rozhraní, která vám umožňují za málo peněz hodně muziky. Špatně definované moduly mají rozsáhlá a složitá rozhraní, která vás nutí používat na jednoduché věci mnoho různých kroků. Dobře definované rozhraní nenabízí příliš mnoho funkcí, na kterých byste záviseli, takže počet vazeb je nízký. Špatně definované rozhraní poskytuje mnoho funkcí, které musíte volat, takže počet vazeb je vysoký.

Dobří softwaroví vývojáři se učí omezovat to, co na rozhraní svých tříd a modulů vystavují. Čím méně má třída metod, tím lépe. Čím méně má funkce proměnných, tím lépe. Čím méně má třída instančních proměnných, tím lépe.

Skrýjte svá data. Skrývejte své pomocné funkce. Skrývejte své konstanty a své dočasné proměnné. Nevytvářejte třídy se spoustou metod nebo spoustou instančních proměnných. Nevytvářejte ve svých podtřídách mnoho chráněných proměnných a funkcí. Soustředte se na to, abyste měli jednolitá a malá rozhraní. Napomáhejte nízkému počtu vazeb tím, že omezíte informace.

## O9: Mrtvý kód

Mrtvý kód je kód, který se neprovádí. Naleznete jej v těle příkazu `i f`, který testuje podmínu, jež nemůže nikdy nastat. Naleznete jej v bloku `catch` příkazu `try`, který nikdy neprovede příkaz `throw`. Naleznete jej v malých pomocných metodách, které se nikdy nevolají, nebo v podmínkách příkazu `switch/case`, jež nikdy nenastanou.

Problém mrtvého kódu spočívá v tom, že se brzy stává nositelem skrytých problémů. Čím je starší, tím silnější a zkaženější má zápach. Je to proto, že mrtvý kód nebyvá při změně návrhu plně aktualizován. Překládá se, ale neřídí se novými konvencemi nebo pravidly. Vznikl v době, kdy byl systém jiný. Když naleznete mrtvý kód, provedte s ním krátký proces. Poskytnete mu slušný pohreb. Vymažte ho ze systému.

## O10: Vertikální oddělování

Proměnné a funkce by měly být definovány poblíž místa, kde se používají. Lokální proměnné by měly být deklarovány těsně před prvním použitím a měly by se používat v malém vertikálním rozsahu. Nechceme deklarovat lokální proměnné stovky řádků daleko od místa, kde se používají.

Soukromé funkce by měly být definovány hned po svém prvním použití. Soukromé funkce patří do rozsahu celé třídy, ale stále bychom rádi omezili vertikální vzdálenost mezi definicí funkce a jejím voláním. Nalézt soukromou funkci by mělo být jen otázkou letmého pohledu dolů od místa prvního použití.

## O11: Nekonzistentnost

Když děláte něco určitým způsobem, dělejte stejně vše ostatní. Tím se vracíme zpátky k principu nejmenšího překvapení. Buděte opatrní, jaké konvence si vyberete, ale pak se jimi pečlivě řídte.

Když použijete v nějaké funkci proměnnou jménem response, do které ukládáte hodnotu typu `HttpServletResponse`, pak používejte konzistentně stejné jméno proměnné i v ostatních funkcích, které používají objekty typu `HttpServletResponse`. Když dáte metodě jméno `processVerificationRequest`, používejte podobné jméno, jako je `processDeletionRequest`, pro metody, které zpracovávají jiné druhy požadavků.

Při vhodném použití takové jednoduché konzistence lze kód lépe modifikovat a bude čitelnější.

## O12: Zaneřáděnost

Jaký význam má implicitní konstruktor, není-li implementován? Neslouží k ničemu jinému, než k zaneřádění kódu nesmyslnými artefakty. Proměnné, které se nepoužívají, funkce, jež se nikdy nevolají, poznámky, které neposkytují žádné informace atd. To vše je jen smetí, které byste měli odstranit. Udržujte své zdrojové kódy čisté, dobře organizované a bez smeti.

## O13: Umělé vazby

Nevytvářejte umělé vazby mezi entitami, které na sobě nezávisejí. Například obecné výčty byste neměli umisťovat do specifickějších tříd, protože to nutí celou aplikaci k tomu, aby tyto třídy používala. Totéž platí pro statické funkce pro všeobecné použití, které jsou deklarovány ve specifických třídách.

Umělá vazba je obecně vazbou mezi dvěma moduly, které neslouží nějakému bezprostřednímu cíli. Je to výsledek ukládání proměnných, konstant nebo funkcí na místo vhodné jen dočasně. Je to výsledek lenosti a nedbalosti.

Věnujte čas a zvažte, kde by se měly funkce, konstanty a proměnné deklarovat. Neházejte je jen tak na nevhodnější místo, které máte po ruce s tím, že je tam pak necháte.

## O14: Chybějící schopnosti

Tento kód je jeden z těch kódů Martina Fowlera, které skrývají nějaký potenciální problém – „zápach v kódu“<sup>6</sup>. Metody nějaké třídy by měly pracovat s proměnnými a funkcemi té třídy, do které patří, a ne

6. [Refactoring].

s proměnnými a funkciemi jiných tříd. Pokud metoda používá přístupové metody a mutátory jiných objektů, aby pracovala s daty v tomto objektu, pak „závidí“ rozsah třídy tohoto objektu. Přeje si, aby byl uvnitř jiné třídy, aby měl přímý přístup k proměnným, se kterými pracuje. Například:

```
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` sahá do objektu `HourlyEmployee`, aby získala data, se kterými dále pracuje. Metoda `calculateWeeklyPay` „závidí“ rozsahu objektu `HourlyEmployee`. „Přeje si“, aby byla uvnitř objektu `HourlyEmployee`.

Ať je to jak chce, je třeba chybějící schopnosti eliminovat, protože to vede k vystavování vnitřních údajů jedné třídy jiné. Avšak někdy jsou chybějící schopnosti nutným zlem. Podívejte se na následující kód:

```
public class HourlyEmployeeReport {  
    private HourlyEmployee employee ;  
    public HourlyEmployeeReport(HourlyEmployee e) {  
        this.employee = e;  
    }  
    String reportHours() {  
        return String.format(  
            "Name: %s\nHours:%d.%ld\n",  
            employee.getName(),  
            employee.getTenthsWorked()/10,  
            employee.getTenthsWorked()%10);  
    }  
}
```

Je vidět, že metoda `reportHours` „závidí“ třídě `HourlyEmployee`. Na druhé straně nechceme, aby měla třída `HourlyEmployee` informace o formátu sestavy. Přesunutí tohoto formátovacího řetězce do třídy `HourlyEmployee` by porušilo několik principů objektově orientovaného návrhu<sup>7</sup>. Vytváří vazbu mezi třídou `HourlyEmployee` a formátem této zprávy a vystavuje ji změnám tohoto formátu.

## O15: Přepínací argumenty

Těžko najdete něco horšího, než argument `false` na konci volání nějaké funkce. Co to má znamenat? Co by se změnilo, kdyby byla jeho hodnota `true`? Nejen že je obtížné si pamatovat účel přepínacího argumentu, ale každý takový argument kombinuje několik funkcí do jedné. Přepínací argumen-

7. Především princip jedné odpovědnosti, princip otevřenosti a uzavřenosti a princip společné uzavřenosti. Viz [PPP].

ty jsou jen výrazem lenosti a snahy zabránit rozdělení velké funkce do několika malých. Podívejte se na tohle:

```
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;
}
```

Argumentem funkce bude hodnota `true`, jsou-li přesčasy placené jedenapůlkrát více, nebo hodnota `false`, jsou-li placené jako běžná pracovní doba. Je nevhodné, abyste si museli pamatovat, co znamená volání `calculateWeeklyPay(false)`, jestliže na ně někdy narazíte. Ale opravdová ostuda takové funkce je, že autor opomněl možnost připsat následující kód:

```
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);
}
```

Samozřejmě že přepínače nemusejí být typu `boolean`. Může jít o výčty, celočíselné hodnoty nebo o jakýkoliv argument, který se používá pro výběr činnosti funkce. Obecně je lepší mít více funkcí než jen jednu, které budete předávat nějaký kód a přepínat mezi jejimi činnostmi.

## O16: Nejasný záměr

Naším přání je, aby byl kód co nejexpresivnější. Výrazy bez mezer, maďarská notace a magická čísla, to vše autorův záměr zatemňuje. Zde je například funkce `overTimePay`, jak mohla vypadat někdy v minulosti:

```
public int m_otCalc() {
    return iThsWkd * iThsRte +
        (int) Math.round(0.5 * iThsRte *
            Math.max(0, iThsWkd - 400)
        );
}
```

Funkce je malá a zahuštěná a je téměř nemožné do ní proniknout. Stojí za to věnovat čas tomu, aby hom zviditelnili náš záměr čtenářům.

## O17: Špatně umístěná odpovědnost

Jednou z nejdůležitějších věcí, jaké může softwarový vývojář udělat, je rozhodnutí, kam umístit kód. Kam by se měla umístit například konstanta `PI`? Měla by být ve třídě `Math`? Možná, že patří do třídy `Trigonometry`. Nebo do třídy `Circle`? Zde přichází v úvahu použití principu nejmenšího překvapení. Kód by měl být umístěn tam, kde by jej čtenář mohl přirozeně očekávat. Konstanta `PI` by měla být tam, kde jsou deklarovány trigonometrické funkce. Konstanta `OVERTIME_RATE` by měla být deklarovaná ve třídě `HourlyPayCalculator`. Někdy se nám ohledně umístění určité funkcionality „rozsvítí“. Dáme ji tam, kde je to pro nás výhodné, ale nemusí to nutně odpovídat intuici čtenáře. Potřebujeme například tisknout sestavu s celkovým počtem hodin, které odpracoval zaměstnanec. Tyto hodiny bychom mohli sečist v kódu, jenž sestavu tiskne, nebo v kódu, ve kterém se zadávají píchací karty.

Jeden způsob, jak dospět k takovému rozhodnutí, je podívat se na názvy funkcí. Řekněme, že modul pro naši sestavu má funkci jménem `getTotalHours`. Řekněme, že tento modul, který přijímá píchačky, obsahuje funkci `saveTimeCard`. Ze které funkce, soudě podle jejího jména, vyplývá, že počítá celkový počet hodin? Odpověď by měla být samozřejmá.

Je jasné, že výkonom může být důvodem, proč by se celkový součet měl počítat v okamžiku zadávání píchaček a ne v době tisku sestavy. To je pěkné, ale názvy funkcí by tomu měly odpovídat. Například v modulu pro píchačky by měla existovat funkce `computeRunningTotalOfHours`.

## O18: Nevhodný modifikátor static

`Math.max(double a, double b)` je dobrá statická metoda. Nepracuje s jednotlivými instancemi. Bylo by skutečně hloupé deklarovat `new Math().max(a, b)` nebo dokonce `a.max(b)`. Veškerá data, která `max` používá, jsou dva argumenty, jež nepocházejí z žádného „vlastnického“ objektu. Kromě toho platí, že zřejmě *nikdy nebudeme potřebovat*, aby tato metoda byla polymorfní.

Někdy však píšeme statické funkce, které by statické být neměly. Například se podívejte na následující kód:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Opakuji, pro tuto funkci je logické, že je `static`. Nepracuje s žádným konkrétním objektem a veškerá data získává prostřednictvím svých argumentů. Existuje však rozumná možnost, že někdy budeme chtít, aby tato funkce byla polymorfní. Můžeme chtít implementovat několik různých algoritmů pro výpočet hodinové mzdy, například `OvertimeHourlyPayCalculator` a `StraightTimeHourlyPayCalculator`. Takže tato funkce by neměla být statická. Měla by být nestatickou členskou funkcí třídy `Employee`.

Obecně platí, že byste měli preferovat nestatické metody před metodami statickými. Jestliže si nejste jisti, deklarujte funkci jako nestatickou. Jestliže opravdu chcete, aby funkce byla statická, přesvědčte se, že v budoucnu nebudeš nikdy potřebovat, aby byla polymorfní.

## O19: Používejte vysvětlující proměnné

O tomto tématu psal Kent Beck v své skvělé knize *Smalltalk Best Practice Patterns*<sup>8</sup> a nedávno znovu ve stejně dobré knize *Implementation Patterns*.

8. [Beck97], str. 108.

Zpřehlednit program je rozdělit výpočty na několik mezivýsledků a ukládat je do proměnných s vysvětlujícími názvy.

Podívejte se na tento příklad 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);
}
```

Jednoduché použití vysvětlujících proměnných jasně sděluje, že první srovnávaná skupina je *key* a druhá je *value*.

Toho není nikdy dost. Obecně je lepší mít více vysvětlujících proměnných než méně. Je zajímavé, jak se může nejasný modul najednou zjasnit, když výpočet jednoduše rozdělíte na mezivýsledky a jejich hodnoty uložíte do dobré pojmenovaných proměnných.

## O20: Názvy funkcí by měly sdělovat, co dělají

Podívejte se na tento kód:

```
Date newDate = date.add(5);
```

Čekali byste, že by tato funkce přičítala pět dnů k datu? Nebo týdnů, či hodin? Mění se instance `date` nebo prostě funkce vrací novou hodnotu `Date`, aniž by měnila starou?

*Na základě volání nemůžete říci, co funkce dělá.*

Jestliže funkce přidává pět dnů k datu a datum mění, měla by se jmenovat `addDaysTo` nebo `increaseByDays`. Jestliže na druhé straně funkce vrací nové datum, které je o pět dnů vyšší, ale nemění datovou instanci, měla by se jmenovat `daysLater` nebo `daysSince`.

Jestliže se musíte podívat na implementaci funkce (nebo do její dokumentace), abyste věděli, co dělá, měli byste pro ni hledat lepší jméno nebo předělat funkcionality tak, aby ji bylo možné umístit do funkcí s lepšími jmény.

## O21: Pochopte algoritmus

Lidé píšou mnoho pochybného kódu, protože si neudělali čas, aby porozuměli algoritmu. Přimějí něco, aby to fungovalo, a vkládají do kódu mnoho příkazů `if` a všelijakých příznaků, aniž by se zastavili a zauvažovali, co se zde vlastně odehrává.

Programování je často zkoumání. *Myslete si, že znáte ten správný algoritmus pro něco, ale pak skončíte tím, že se v kódu štáráte, rýpete a hrajete si s tím, až to konečně „funguje“.* Jak víte, že to „funguje“? Protože to projde všemi testy, na které si jen pomyslíte. Na tomto přístupu není nic špatného. Je pravda, že je to často jediná možnost, jak donutit funkci dělat to, o čem jste přesvědčeni, že by měla dělat. Avšak nestačí, že u slova „funguje“ odstraníte pouze jeho uvozovky.

Než si začnete myslit, že jste s funkcí hotovi, prověrte si, že *rozumíte* tomu, jak funguje. Nestačí, že projde všemi testy. Musíte *vědět*, že řešení je korektní.

- 
9. Je rozdíl v tom, zda víte, jak pracuje kód nebo zda bude algoritmus dělat to, co od něj požadujeme. Nejistota, zda je algoritmus v pořádku, je v životě častým jevem. Nejistota, co dělá váš kód, je pouze otázkou lenosti.

Často je tou nejlepší metodou, jak pochopit algoritmus, refaktorovat funkci na něco, co je dostatečně čisté a natolik expresivní, že její princip bude naprosto evidentní.

## O22: Udělejte z logických závislostí fyzické

Jestliže jeden modul závisí na jiném, měla by být tato závislost fyzická, ne pouze logická. Závislý modul by neměl dělat předpoklady o modulu, na kterém závisí (jinými slovy, neměl by vytvářet logické závislosti). Místo toho by tento modul měl explicitně žádat o veškeré informace, na kterých závisí.

Představte si například, že píšete funkci, která tiskne jednoduchou textovou zprávu o odpracovaných hodinách zaměstnanců. Jedna třída jménem HourlyReporter shromažďuje veškerá data do vhodného formátu a ta pak předá objektu typu HourlyReportFormatter, aby je vytiskl (viz výpis 17.1).

### Výpis 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)
                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;
    }
}
```

Tento kód obsahuje logickou závislost, která nemá fyzickou podobu. Všimli jste si ji? Je to konstanta PAGE\_SIZE. Proč by měla třída HourlyReporter obsahovat informace o velikosti stránky? Velikost stránky by měla být obsažena v odpovědnosti objektu typu HourlyReportFormatter.

Skutečnost, že je konstanta PAGE\_SIZE deklarována ve třídě HourlyReporter, znamená, že jede o špatně umístěnou odpovědnost [O17]. Ta způsobuje, že třída HourlyReporter předpokládá, že zná velikost, jakou by stránka měla mít. Takový předpoklad znamená logickou závislost. Třída HourlyReporter závisí na skutečnosti, že objekt typu HourlyReportFormatter umí zpracovat stránky o délce 55. Kdyby nějaká implementace třídy HourlyReportFormatter nedokázala tuto délku zpracovat, došlo by k chybě.

Tuto závislost můžeme změnit na fyzickou, když v objektu HourlyReportFormatter vytvoříme novou metodu jménem getMaxPageSize(). Tu pak může volat třída HourlyReporter, místo aby používala konstantu PAGE\_SIZE.

## O23: Volte raději polymorfismus než příkazy if/else nebo switch/case

Tento návrh může být zdánlivě v rozporu s tématem kapitoly 6. V této kapitole jsem dospěl k závěru, že příkazy switch jsou pravděpodobně nevhodné v těch částech systému, kde budeme spíše přidávat nové funkce než nové typy.

Za prvé, většina lidí používá příkazy switch, protože vedou k řešení problému hrubou silou, a ne proto, že by se jednalo pro danou situaci o správné řešení. Takže tato heuristika nám zde má připomenout, abychom zvážili, zda bychom neměli dát přednost polymorfismu před příkazem switch.

Za druhé, případy, kdy se přidávají funkce a ne typy, jsou relativně řídké. Takže každý příkaz switch by měl být podezřelý.

Já sám dodržuju následující pravidlo „jednoho příkazu switch“: *V daném výběru by neměl být více než jeden příkaz switch. Větve case příkazu switch musí vytvořit polymorfní objekty, které nahrazují ostatní příkazy switch ve zbyvající části systému.*

## O24: Dodržujte standardní konvence

Každý tým by měl dodržovat normy pro psaní kódu, který je založen na všeobecných normách daného odvětví. Tento standard by měl specifikovat věci, jako kde deklarovat instanční proměnné, jak pojmenovat třídy, metody a proměnné, kam umístit závorky atd. K popisu těchto konvencí by tým neměl potřebovat dokumentaci, protože samotný kód je toho příkladem.

Každý člen týmu by se měl těmito konvencemi řídit. To znamená, že každý člen týmu musí být nataknut vyspělý, aby pochopil, že nezáleží na tom, kam budete umisťovat své závorky, pokud se na jejich umístění všichni shodnete.

Kdybyste chtěli vědět, kterým konvencím dávám přednost já, podívejte se na refaktorovaný kód na straně 402 ve výpisech B.7 až B.14.

## O25: Nahradte magická čísla pojmenovanými konstantami

Tohle je pravděpodobně nejstarší pravidlo v historii vývoje softwaru. Vzpomínám si, že jsem o něm četl v druhé polovině šedesátých let v úvodu manuálů pro jazyk COBOL, FORTRAN a PL/I. Obecně platí, že není dobré, aby kód používal čísla v surovém tvaru. Měli byste je deklarovat jako konstanty a dobré je pojmenovat.

Například číslo 86400 by mělo být deklarované jako konstanta `SECONDS_PER_DAY`. Pokud tisknete 55 řádek na stránku, pak by se konstanta 55 měla deklarovat jako `LINES_PER_PAGE`.

Některé konstanty jsou natolik zřejmé, že není třeba jim dávat jméno, pokud se používají ve spojení s dobré vysvětlujícím kódem. Například:

```
double milesWalked = feetWalked/5280.0;  
int dailyPay = hourlyRate * 8;  
double circumference = radius * Math.PI * 2;
```

Potrebujeme skutečně v uvedených příkladech konstanty `FEET_PER_MILE`, `WORK_HOURS_PER_DAY` a `TWO`? Použití té poslední je určitě absurdní. Pro některé vzorce je lepší, když se konstanty používají v původní číselné podobě. Mohli bychom jistě diskutovat ohledně konstanty `WORK_HOURS_PER_DAY`, protože se pravidla konvence mohou měnit. Na druhé straně je tento vzorec s číslem 8 celkem čitelný, a proto bych se zdráhal přidávat dalších 17 znaků a zatežovat tím čtenáře. A konstanta `FEET_PER_MILE`, pro číslo 5280, které je tak dobré známé (jak kde – pozn. odborného korektora), že by je čtenář pochopil, i kdyby bylo na stránce samotné bez jakéhokoliv doprovodného kontextu.

Konstanty, jako je 3,141592653589793, jsou rovněž dobré známé a snadno rozpoznatelné. Avšak riziko chyby je příliš velké, než abychom ji používali v její číselné podobě. Pokaždé, když někdo uvidí číslo 3.1415927535890793, tak ví, že se jedná o  $\pi$  a určitě jej nebude kontrolovat. (Všimli jste si chyb v některých číslicích?) Rovněž není žádoucí, aby lidé používali čísla jako 3,14, 3,14159, 3,142 atd. Je tedy dobré, že konstanta `Math.PI` je pro nás již definovaná.

Termín „magické číslo“ se netýká jen čísel. Týká se jakéhokoliv symbolu, který reprezentuje nějakou hodnotu a sám není dostatečně srozumitelný. Například:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

V této aserci jsou dvě magická čísla. Prvním z nich je samozřejmě 7777, i když není jasné, co znamená. Druhé magické číslo je „John Doe“, a opět zde není jasný záměr.

Ukazuje se, že „John Doe“ je jméno zaměstnance s číslem 7777 ve známé testovací databázi, kterou vytvořil náš tým. Každý člen našeho týmu ví, že když se k této databázi připojí, bude tam několik připravených zaměstnanců se známými hodnotami a atributy. Rovněž se ukazuje, že „John Doe“ reprezentuje v testovací databázi jediného zaměstnance, který je placený hodinově. Takže test byste měli číst takto:

```
assertEquals(  
    HOURS_EMPLOYEE_ID,  
    Employee.find(HOURS_EMPLOYEE_NAME).employeeNumber());
```

## O26: Budě přesní

Byla by naivní předpokládat, že první záznam výběru bude záznamem *jediným*. Používat čísla v po-hyblivé řádové čárce pro měnu je téměř kriminální čin. Vyhýbání se zámkkům nebo transakcím jen proto, že si myslíte, že nemůžete dojít k souběžné aktualizaci, je přinejmenším lenost. Deklarovat proměnnou typu `ArrayList`, když by stačil typ `List`, je příliš omezující. Mít všechny proměnné implicitně jako `protected` zase neomezuje dostatečně.

Když provedete nějaké rozhodnutí ohledně kódu, tak ho proveďte precizně. Měli byste vědět, proč jste ho udělali a jak se budete potýkat s jakoukoli výjimkou. Nebudete líní ohledně detailů svých rozhodnutí. Pokud se rozhodnete volat funkci, která by mohla vracet hodnotu `null`, ověrte si, že výsledek budete kontrolovat. Když hledáte záznam, o kterém si myslíte, že je v databázi jediný, ověrte si, že tam nejsou žádné jiné. Když bude pracovat s měnou, používejte čísla typu `integer`<sup>10</sup> a vyřešte vhodným způsobem zaokrouhlování. Jestliže existuje možnost souběžné aktualizace, použijte v každém případě nějaký zamykací mechanismus.

Dvojznačnost a nepřesnosti v kódu jsou výsledkem buď neshod, nebo lenosti. Měly by být v každém případě odstraněny.

## O27: Struktura je více než konvence

Dejte přednost rozhodnutím ohledně návrhu před pravidly konvence. Konvence pro tvorbu jmen je dobrá, ale není tak důležitá jako struktury, které musí vyhovovat daným pravidlům. Například příkazy `switch/cases` s dobře pojmenovanými výčty nejsou tak důležité jako základní třídy s abstraktními metodami. Nikdo není povinen implementovat příkaz `switch/case` vždy stejným způsobem, ale základní třídy nás nutí k tomu, aby konkrétní třídy měly implementovány všechny abstraktní metody.

## O28: Zapouzdřete podmínky

Kód s logickými příkazy je obtížně srozumitelný, a musíte-li studovat kontext příkazů `if` nebo `while`, je to ještě horší. Extrahujte funkce, které vysvětlují záměr logických výrazů. Například:

```
if (shouldBeDeleted(timer))

je lepší než

if (timer.hasExpired() && !timer.isRecurrent())
```

## O29: Vyhýbejte se negativním podmíněným výrazům

Negativní podmíněné výrazy jsou o něco méně srozumitelné než pozitivní. Takže vytvářejte pozitivní podmíněné příkazy všude, kde je to možné. Například:

```
if (buffer.shouldCompact())

je lepší než

if (!buffer.shouldNotCompact())
```

10. Nebo ještě lépe, třídu `Money`, která používá celá čísla.

## O30: Funkce by měly provádět jen jednu věc

Často je lákavé vytvářet funkce s mnoha sekčemi, které provádějí více operací. Funkce tohoto druhu dělají více než jednu věc, a proto by se měly rozdělit na menší funkce, z nichž by každá dělala jen jednu věc. Například:

```
public void pay() {
    for (Employee e : employees) {
        if (e.isPayday()) {
            Money pay = e.calculatePay();
            e.deliverPay(pay);
        }
    }
}
```

Tento krátký úsek kódu provádí tři věci. Prochází cyklicky zaměstnance, testuje, zda by měl dostávat mzdu, a tuto mzdu počítá. Tato funkce vy vypadala lépe takto:

```
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);
}
```

Každá z těchto funkcí dělá jen jednu věc. (Viz „Dělejte jen jednu věc“ na straně 57.)

## O31: Skrytá časová vazba

Časové vazby jsou někdy nevyhnutelné, ale neměli byste je skrývat. Uspořádejte argumenty vaší funkce tak, aby pořadí jejich předávání bylo samozřejmé. Podívejte se na následující kód:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;
    public void dive(String reason) {
        saturateGradient();
        reticulateSplines();
        diveForMoog(reason);
    }
    ...
}
```

U volání těchto tří funkcí je důležité jejich pořadí. Nejdříve musíte saturovat gradient, pak můžete sítí pokrýt křivky a teprve poté můžete volat poslední funkci. Ze samotného kódu není bohužel patrná

nutnost této časové posloupnosti. Jiný programátor by mohl volat metodu `reticulateSplines` před metodou `saturateGradient`, což by vedlo k výjimce typu `UnsaturatedGradientException`.

Zde je lepší řešení:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;
    public void dive(String reason) {
        Gradient gradient = saturateGradient();
        List<Spline> splines = reticulateSplines(gradient);
        diveForMoog(splines, reason);
    }
    ...
}
```

To nám ukazuje časovou vazbu tím, že vytvoříme „kaskádu věder“. Každá funkce vrací výsledek, který je potřeba pro následující funkci. Nemáme tedy žádný rozumný důvod je volat v jiném pořadí.

Můžete namítat, že tento postup zvyšuje složitost funkcí, a máte pravdu. Ale syntaktická složitost navíc ukazuje skutečnou časovou složitost okolností.

Všimněte si, že jsem v kódu ponechal instanční proměnné. Předpokládám, že je budou potřebovat soukromé metody třídy. Přesto si přejí, aby použité argumenty tuto časovou vazbu vyjadřovaly.

## O32: Nepodléhejte libovůli

Zdůvodněte si strukturu svého kódu a přesvědčte se, že tyto důvody jsou v jeho struktuře skutečně vidět. Jestliže vypadá struktura kódu jako něco libovolného, budou mít jiní nutkání ji měnit. Jestliže vypadá struktura v rámci celého systému konzistentně, budou ji ostatní používat a budou zachovávat konvence. Nedávno jsem například slučoval změny ve FitNesse a zjistil jsem, že jeden z přispěvatelů provedl tohle:

```
public class AliasLinkWidget extends ParentWidget
{
    public static class VariableExpandingWidgetRoot {
        ...
    }
}
```

Problém spočívá v tom, že není třeba, aby třída `VariableExpandingWidgetRoot` byla v rozsahu třídy `AliasLinkWidget`. Dále, ostatní nesouvisející třídy používají třídu `AliasLinkWidget`. `VariableExpandingWidgetRoot`. Tyto třídy nepotřebují o třídě `AliasLinkWidget` vůbec vědět.

Možná, že programátor umístil třídu `VariableExpandingWidgetRoot` do třídy `AliasWidget` jen kvůli pohodlnosti, nebo si myslí, že tam opravdu má být. Ať už byl důvod jakýkoliv, výsledný dojem je, jako by šlo o jeho libovůli. Veřejné třídy, které nejsou pomocnými nástroji nějakých jiných tříd, by neměly být umístěny uvnitř jiných tříd. Pravidlem je, aby byly veřejné a na nejvyšší úrovni svého balíčku.

## O33: Zapouzdřete hraniční podmínky

Hraniční podmínky se velmi obtížně sledují. Vložte kód pro jejich zpracování na jedno místo. Nenechte je, aby prosákly celým vaším kódem. Nepotřebujeme spoustu plus jedniček a minus jedniček a není nutné, aby byly roztroušené tady a zase tamhle. Podívejte se na jednoduchý příklad z FIT:

```
if(level + 1 < tags.length)
{
    parts = new Parse(body, tags, level + 1, offset + endTag);
    body = null;
}
```

Všimněte si, že se výraz `level+1` objevuje dvakrát. Jedná se o hraniční podmínu, která by měla být zapouzdřena do proměnné jménem `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
    parts = new Parse(body, tags, nextLevel, offset + endTag);
    body = null;
}
```

## O34: Funkce by měly sestupovat jen o jednu úroveň abstrakce níže

Příkazy ve funkci by měly být všechny na stejně úrovni abstrakce, která by měla být o jednu úroveň níže, než je operace, již popisuje název funkce. Vysvětlit tuto heuristiku a také se jí řídit, to je jedním z nejtěžších úkolů. I když je tato myšlenka dostatečně jasná, lidé umějí smíchat různé úrovně abstrakce opravdu dobře. Podívejte se například na následující kód 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();
}
```

Stačí chvílička a již víte, o co tu jde. Tato funkce vytváří značku HTML, která na stránce nakreslí vodorovnou čáru. Tloušťku čáry stanovuje proměnná `size`.

Podívejte se nyní ještě jednou. Tato metoda směšuje minimálně dvě úrovně abstrakce. První z nich je představa, že vodorovná čára má velikost. Druhou je syntaxe vlastní značky HR. Tento kód pochází z modulu HruleWidget z FitNesse. Detekuje řádek složený ze čtyř nebo více pomlček a převádí je na odpovídající značku HR. Čím více pomlček, tím je větší `size`.

Jak uvidíte dále, tento kód jsem trochu refaktoroval. Všimněte si, že jsem změnil jméno složky `size`, abych vyjádřil jeho skutečný smysl. Pole obsahovalo počet pomlček, které byly navíc.

```

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);
}

```

Tato změna celkem dobře odděluje dvě úrovně abstrakce. Funkce `render` vytváří značku HR, aniž by věděla cokoliv o syntaxi HTML. Všechny nepříjemné otázky syntaxe HTML řeší modul `HtmlTag`.

Po této změně jsem objevil drobnou chybu. Původní kód neumisťoval do značky HR uzavírací lomítko, jak to vyžaduje standard XHTML. (Jinými slovy generuje `<hr>` místo `<hr/>`.) Modul `HtmlTag` se změnil již dříve, aby vyhovoval standardu XHTML.

Oddělení úrovní abstrakce je jednou z nejdůležitějších funkcí refaktorování a je velmi obtížné to provést dobře. Podívejte se například na následující kód. To byl můj první pokus oddělit úrovně abstrakce v metodě `HruleWidget.render`.

```

public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (size > 0) {
        hr.addAttribute("size", ""+(size+1));
    }
    return hr.html();
}

```

V tomto okamžiku bylo mým cílem provést nezbytnou separaci a zajistit, aby prošly testy. Toho jsem dosáhl celkem jednoduše, ale výsledkem byla funkce, jejíž úrovně abstrakce byly *stále* pomíchané. V našem případě tyto smíchané úrovně abstrakce tvořily konstrukce značky HR, interpretace a formátování proměnné `size`. To vede k závěru, že když funkci rozdělíte podle úrovní abstrakce, často odhalíte nové úrovně abstrakce, které nebyly díky předchozí struktuře zřetelné.

## O35: Mějte konfigurační data na vysokých úrovních

Jestliže máte konstanty, jako jsou implicitní nebo konfigurační hodnoty, jež jsou známé, a u kterých očekáváme, že budou ve vyšších úrovních abstrakce, neskrývejte je v nižších úrovních. Odhalte je jako argumenty funkce nízké úrovně, která se předává z funkce vyšší úrovně. Podívejte se na následující kód 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 příkazového řádku jsou analyzovány ve FitNesse na prvním proveditelném řádku. Výchozí hodnoty těchto argumentů jsou specifikovány na začátku třídy Argument. Nemusíte se chodit dívat do nižších úrovní systému na příkazy, jako je tento:

```
if (arguments.port == 0) // implicitně použij 80
```

Konfigurační konstanty jsou umístěny na nejvyšší úrovni a je jednoduché je měnit. Pak se předávají níže ostatním částem aplikace. Nejnižší úrovně aplikace hodnoty konstant neobsahují.

## O36: Vyhnete se tranzitivním odkazům

Obecně nechceme, aby nějaký modul měl příliš mnoho informací o svých spolupracovnících. Přesněji řečeno, jestliže modul A spolupracuje s modulem B a B spolupracuje s C, nechceme, aby moduly, které používají A, věděly o modulu C. (Nechceme například volat metodu, jako je a.getB().getC().doSomething().) Někdy tomu říkáme Démétrin zákon. Pragmatictí programátoři tomu říkají „psát nesmělý kód“<sup>11</sup>. V každém případě z toho plyne závěr, že je třeba zajistit, aby moduly věděly jen o svých bezprostředních spolupracovnících a neznaly cestovní mapu celého systému.

Jestliže příkaz a.getB().getC() používá více modulů, bude obtížné změnit návrh a architekturu tak, abychom vložili Q mezi B a C. Museli byste vyhledat každé volání tvaru a.getB().getC() a změnit ho na a.getB().getQ().getC(). Takto architektura rychle kostnatí. Příliš mnoho modulů má informace o architektuře systému.

Chceme, aby naše bezprostředně spolupracující moduly raději nabízely veškeré služby, které potřebujeme. Neměli bychom se potulovat diagramem objektů systému a hledat metodu, kterou chceme volat. Spíše bychom měli být schopni říci:

```
myCollaborator.doSomething().
```

## Java

### J1: Vyhnete se dlouhým seznamům a používejte zástupné znaky

Jestliže používáte dvě nebo více tříd z nějakého balíčku, importujte celý balíček pomocí příkazu

```
import package.*;
```

Pro čtenáře jsou dlouhé seznamy importovaných souborů břemenem. Nechceme zaneřádit začátek modulů osmdesáti rádky importů. Spiše bychom chtěli, aby byly importované soubory definovány stručným příkazem s uvedením balíčku, se kterým budeme pracovat.

Přesné stanovené importy jsou, na rozdíl od zástupných znaků, komplikované závislosti. Když importujete přesně definovanou třídu, musí tato třída *existovat*. Když ale importujete balíček pomocí zástupných znaků, nemusí existovat žádná konkrétní třída. Příkaz k importu prostě přidá balíček do seznamu cest, které se budou při doplňování jmen prohledávat. Takže tímto způsobem nevzniká žádná reálná závislost, címž vzniká u našich modulů méně vazeb.

Někdy mohou být dlouhé seznamy přesně stanovených importovaných souborů užitečné. Například když pracujete se zděděným kódem a chcete zjistit, které třídy potřebujete k sestavení imitátorů a objektů s prázdným kódem, můžete procházet seznamy specifických importovaných souborů a pro veškeré třídy hledat existující kvalifikovaná jména, za něž vložíte do kódu adekvátní objekty s prázdným kódem. Avšak takové použití přesně definovaných importů je velmi neobvyklé. Dále platí, že většina moderních vývojových prostředí vám jediným příkazem umožní konvertovat zástupné znaky na seznam přesně definovaných importů. Takže používání zástupných znaků je lepší i v případě zděděného kódu.

Někdy způsobí importy pomocí zástupných znaků dvojznačnosti a konflikty mezi jmény. Dvě třídy se stejným jménem, ale v různých balíčcích, vyžadují zvláštní druh importu nebo alespoň kvalifikovaný odkaz. Tohle může být otravné, ale stává se to velmi zřídka, takže používání zástupných znaků je obecně stále lepší, než přesně specifikovaný import.

## J2: Vyhñe se dědění konstant

S děděním konstant jsem se setkal již několikrát a vždy jsem u toho protáhl obličeji. Programátor umístí nějaké konstanty do rozhraní a pak k nim získá přístup pomocí dědění. Podívejte se na následující kód:

```
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)
        );
    }
    ...
}
```

Kde se vzaly konstanty `TENTHS_PER_WEEK` nebo `OVERTIME_RATE`? Mohly by být třeba ve třídě `Employee`, a tak se tam pojďme podívat:

```
public abstract class Employee implements PayrollConstants {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}
```

Ne, tam nejsou. Kde tedy? Podívejte se podrobněji na třídu Employee. Implementuje rozhraní PayrollConstants.

```
public interface PayrollConstants {  
    public static final int TENTHS_PER_WEEK = 400;  
    public static final double OVERTIME_RATE = 1.5;  
}
```

To je ale hrozný zvyk! Konstanty jsou skryté na vrcholu dědické hierarchie. Fuj! Nepoužívejte dědění, abyste obelstili jazyková pravidla pro rozsahy platnosti. Místo toho použijte statický import.

```
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: Konstanty versus výčty

Nyní, když Java 5 obsahuje i výčty, používejte je! Přestaňte používat starý trik public static final int. Význam položek typu int se může vytratit. Význam výčtových konstant nikoliv, protože jsou součástí výčtu, který je pojmenovaný.

Dále se pečlivě podívejte na syntaxi výčtu. Mohou obsahovat metody i pole. To z nich dělá velmi výkonné nástroje, které poskytují mnohem větší expresivitu a pružnost, než celočíselné položky. Podívejte se na jednu variantu kódu pro výpočet výplat:

```
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();
}

```

## Jména

### Jm1: Vybírejte popisná jména

Při výběru jmen nikam nespěchejte. Snažte se, ať jsou popisná. Nezapomínejte, že významy jmen se mění spolu s vývojem softwaru, takže neváhejte častěji přehodnotit vhodnost jmen, která jste vybrali.

Tohle není jen nějaké doporučení pro dobrý pocit. Jména jsou v softwaru z devadesáti procent tím, co jej činí čitelným. Rozumný výběr jmen chce svůj čas, aby zůstávala relevantní. Jsou příliš důležitá na to, abyste s nimi zacházeli neopatrně.

Podívejte se na následující kód. Co dělá? Kdybych vám tento kód ukázal s dobře volenými jmény, pak byste jeho smysl pochopili, ale takto se bude jevit jako směsice symbolů a magických čísel.

```

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;
}

```

Následuje kód ve tvaru, jak by měl správně vypadat. Tato část kódu je ve srovnání s předchozím kódem neúplná. Můžete z ní ihned dedukovat, co se pokouší dělat, a na základě těchto dedukcí budete pravděpodobně umět chybějící funkce dopsat. Magická čísla již nejsou magická a struktura algoritmu je přesvědčivá a popisná.

```
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;
}
```

Význam pečlivě zvolených jmen spočívá v tom, že zaplní strukturu kódu popisnými údaji. Jejich nadbytečnost vybudí u čtenáře očekávání, k čemu mohou být ostatní funkce v modulu. Na základě výše uvedeného kódu můžete usuzovat, jak bude vypadat implementace metody `isStrike()`. Když budete tuto metodu studovat, bude tím, „co jste do značné míry předpokládali“<sup>12</sup>.

```
private boolean isStrike(int frame) {
    return rolls[frame] == 10;
}
```

## Jm2: Vybírejte jména na adekvátní úrovni abstrakce

Nevolte jména, která informují o implementaci. Volte taková, která odrážejí úroveň abstrakce třídy nebo funkce, s níž pracujete. To není jednoduché. Opakuj, že lidé jsou v míchání úrovni abstrakce velmi dobrí. Pokaždé, když si projdete svůj kód, asi najeznete nějaké proměnné, jejichž jméno odpovídá příliš nízké úrovni. Měli byste využít okamžiku, když takové proměnné najeznete, a změnit jejich jména. Udělat kód čitelnějším vyžaduje, abyste se věnovali jeho nepřetržitému zdokonalování. Podívejte se na rozhraní Modem:

```
public interface Modem {
    boolean dial(String phoneNumber);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedPhoneNumber();
}
```

12. Viz citát Warda Cunninghama na straně 34.

Na první pohled to vypadá hezky. Všechny funkce se zdají být v pořádku. Je pravda, že u mnoha aplikací tomu tak je. Ale podívejte se na aplikaci, v níž některé modemy nejsou připojeny pomocí vytáčeného spojení. Jsou permanentně připojeny na společnou pevnou linku (viz kabelové modemy, které zabezpečují přístup k Internetu ve většině dnešních domácností). Možná, že některé jsou připojeny pomocí přepínače přes USB, kterému posílají číslo portu. Myšlenka telefonních čísel je zřejmě na špatné úrovni abstrakce. V tomto scénáři bychom k tvorbě jmen mohli přistupovat lépe:

```
public interface Modem {
    boolean connect(String connectionLocator);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedLocator();
}
```

Pokud jde o telefonní čísla, ze samotných jmen nelze nic vyvzakovat. Lze je použít jak pro telefonní čísla, tak pro jakýkoliv jiný druh spojení.

### **Jm3: Používejte standardní názvosloví všude, kde je to možné**

Je snadnější porozumět jménům, tvoří-li se na základě existujících pravidel nebo předchozí praxe. Pokud používáte například vzor dekorátor (DECORATOR), měli byste používat slovo *Decorator* ve jménech dekorujících tříd. Například *AutoHangupModemDecorator* by mohlo být jméno třídy, která dekoruje *Modem* s možností automatického zavěšení na konci relace.

Vzory jsou jen jedním druhem standardu. V Javě se například funkce, které převádějí objekty na řetězce, často jmenují *toString*. Je lepší dodržovat takovéto konvence než vynalézat vlastní.

Týmy často vytvářejí své standardy pro tvorbu jmen určené jen pro specifický projekt. Eric Evans tomu říká *všudypřítomný jazyk* projektu<sup>13</sup>. Váš kód by měl v maximální míře používat termíny tohoto jazyka. Stručně řečeno, čím více můžete používat jmen, která jsou nositelj zvláštních významů, relevantních pro váš projekt, tím jednodušejí budou čtenáři váš kód číst.

### **Jm4: Jednoznačná jména**

Vybírejte jména, která jednoznačně popisují činnost funkce nebo proměnné. Podívejte se na následující příklad z FitNesse:

```
private String doRename() throws Exception
{
    if(refactorReferences)
        renameReferences();
    renamePage();
    pathToRename.removeNameFromEnd();
    pathToRename.addNameToEnd(newName);
    return PathParser.render(pathToRename);
}
```

13. [DDD].

Jméno této funkce se vyjadřuje pomocí obecných a vágních termínů, které neříkají nic o tom, co funkce dělá. To ještě zdůrazňuje skutečnost, že uvnitř funkce `doRename` je funkce jménem `renamePage`! Co vám říkají tato jména o rozdílech mezi oběma funkciemi? Nic.

Lepší jméno pro tuto funkci je `renamePageAndOptionallyAllReferences`. Může se vám zdát dlouhé, a dlouhé to skutečně je, ale k volání dochází na jediném místě v tomto modulu, takže jeho vysvětlující hodnota vyvažuje jeho délku.

## Jm5: Pro velké rozsahy používejte dlouhá jména

Délka jména by měla mít nějaký vztah k délce rozsahu. Pro malé rozsahy můžete používat velmi krátká jména, ale pro velké rozsahy byste měli použít delší jména.

Proměnné, jako jsou `i` nebo `j`, jsou dobré pro rozsah o délce do pěti řádků. Podívejte se na tento kousek kódu ze staré klasické hry kuželky:

```
private void rollMany(int n, int pins)
{
    for (int i=0; i<n; i++)
        g.roll(pins);
}
```

To je zcela jasné a význam tohoto kódu by byl méně srozumitelný, pokud bychom místo proměnné `i` použili něco složitějšího, jako například `rollCount`. Na druhé straně proměnné a funkce s krátkými jmény po nějakém počtu řádků ztrácejí svůj význam. Takže cím je rozsah větší, tím by mělo být jméno delší a přesnější.

## Jm6: Vyhnete se kódování jmen

Jména by neměla obsahovat kódy s informacemi o typu nebo rozsahu. Předpony, jako jsou `m_` nebo `f`, jsou v dnešních vývojových prostředích zbytečné. Také kódy, obsahující informace o projektu nebo o subsystému, jako je `vis_` (pro vizuální zobrazovací systém), jsou nadbytečné a rozptýlují pozornost. Opět platí, že dnešní vývojová prostředí takové informace poskytuje a nemusíte proto předělávat jména. Nepoužívejte názvy s maďarskou notací.

## Jm7: Jména by měla popisovat vedlejší efekty

Jména by měla popisovat vše, čím funkce, proměnná nebo třída je nebo co dělá. Neskrývejte za jméno vedlejší efekty. Nepoužívejte jednoduchá slovesa, abyste popsali funkci, která dělá více než jen jednoduchou činnost. Podívejte se například na následující kód z testovací šablony TestNG:

```
public ObjectOutputStream getOos() throws IOException {
    if (m_oos == null) {
        m_oos = new ObjectOutputStream(m_socket.getOutputStream());
    }
    return m_oos;
}
```

Tato funkce dělá něco více, než že získává jen „oos“. Funkce vytvoří „oos“, pokud ještě vytvořen není. Z tohoto důvodu by mohlo být lepší jméno `createOrReturnOos`.

# Testy

## T1: Nedostatečné testy

Kolik by mělo být testů v testovací sadě? Bohužel, míra, kterou používá většina programátorů, je: „Tohle by mohlo stačit“. Testovací sada by měla otestovat vše, co by nemuselo eventuálně fungovat. Testy nejsou úplné, když někde existují podmínky, které dosud nebyly těmito testy otestovány, nebo výpočty, jež neprošly jejich kontrolou.

## T2: Používejte nástroje pro pokrytí

Nástroje pro pokrytí kódu vám odhalí mezery ve vaší testovací strategii. Zjednoduší výhledávání modulů, tříd a funkcí, které netestujete dostatečně. Většina integrovaných vývojových prostředí vám poskytne vizuální náhled a označí zeleně řádky, které jsou pokryty testy, a červeně ty, jež pokryty nejsou. To umožňuje nalézt rychle a jednoduše příkazy `if` nebo `catch`, jejichž těla dosud neprošla kontrolou.

## T3: Nepreskakujte triviální testy

Jsou velmi jednoduché a jejich dokumentační hodnota je vyšší, než jejich pořizovací náklady.

## T4: Opomenutý test je otázkou ohledně nejednoznačnosti

Někdy si nejsme jisti, jak má nějaká dílká funkčnost vypadat, protože není zcela jasné zadání. Pochybnosti ohledně zadání můžeme vyjádřit pomocí zakomentovaných testů nebo testů označených anotací `@Ignore`. Volit musíte podle toho, aby tyto nejasnosti nezpůsobily problémy s překladem.

## T5: Testujte hraniční podmínky

Věnujte zvláštní pozornost testování hraničních podmínek. Často je jádro algoritmu správně, ale špatně zhodnotíme hraniční podmínky.

## T6: V okolí programových chyb provádějte důkladné testy

Programové chyby mají tendenci se shlukovat. Když naleznete ve funkci chybu, je rozumné podrobit tuto funkci důkladným testům. Pravděpodobně zjistíte, že tato chyba nebyla jediná.

## T7: Zákonitosti v selhávání odhalují chyby

Někdy můžete určit problém tak, že ve způsobu selhávání testů naleznete nějaké zákonitosti. Je to jen další důvod, proč mít testy co nejúplnější. Kompletní testy, spouštěné ve správném pořadí, mohou tyto zákonitosti odhalit.

Předpokládejme, že jste si všimli, že všechny testy selhaly, pokud je vstupní řetězec delší než pět znaků. Nebo že selhal každý test, který předal jako druhý argument nějaké funkce záporné číslo. Někdy se stáčí jen podívat, které řádky jsou červené a jaké zelené, a můžeme zvolat „Aha!“, protože jsme nalezli řešení. Podívejte se zpět na zajímavý příklad (strana 274), uvedený během rozboru třídy `SerialDate`.

## T8: Pokrytí kódu testy může odhalit chyby

Když se podíváte na kód, jenž není otestován a neprovádí se, můžete nalézt stopy vedoucí k příčinám, proč některé testy selhávají.

## T9: Testy by měly být rychlé

Pomalé testy nebude chtít nikdo spouštět. Když půjde do tuhého, v prvé řadě to budou pomalé testy, které budou z testovací sady odstraněny. Takže *udělejte vše nezbytné* pro to, aby byly testy rychlé.

# Závěr

Nelze říci, že by tento seznam heuristik a potenciálních problémů byl úplný. A opravdu si nejsem jist, zda je *vůbec možné*, aby podobný seznam úplný byl. Ale možná, že by cílem neměla být jeho úplnost, protože tento seznam má vést především k nějakému systému hodnot.

Hodnotový systém je skutečně hlavním cílem a tématem této knihy. Čistý kód nevznikne dodržováním nějakých pravidel. Dobrým programátorem se nestanete tím, že se naučíte seznam heuristik. Profesionalismus a um vycházejí z hodnot, které jsou hybnou silou správných postupů.

## Použitá literatura

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

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

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

[Beck97]: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

[Beck07]: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.

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

[DDD]: *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

# DODATEK A

## Souběžnost II

*Brett L. Schuchert*

### **V této kapitole najdete:**

- ◆ Příklad klient/server
- ◆ Počet cest při provádění kódu
- ◆ Vyznejte se ve své knihovně
- ◆ Závislost mezi metodami může souběžný kód porušit
- ◆ Zvyšování propustnosti
- ◆ Zablokování
- ◆ Testování kódu s více podprocesy
- ◆ Podpora nástrojů pro testování kódu s podprocesy
- ◆ Závěr
- ◆ Výukový program



Tento dodatek doplňuje kapitolu *Souběžnost* na straně 189. Tvoří ji několik nezávislých témat a můžete je číst v libovolném pořadí. Někde se text opakuje, aby takové čtení bylo možné.

## Příklad klient/server

Představte si jednoduchou aplikaci typu klient/server. Server poslouchá prostřednictvím soketu a čeká na požadavek klienta, aby navázaly spojení. Klient spojení naváže a vyšle požadavek.

### Server

Zde je zjednodušená verze serverové aplikace. Úplný zdrojový kód tohoto příkladu je k dispozici na straně 345 s názvem *Klient/server bez podprocesů*.

```
ServerSocket serverSocket = new ServerSocket(8009);
while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Tato jednoduchá aplikace čeká na spojení, zpracuje příchozí zprávu a opět čeká na další příchozí požadavek klienta. Zde je kód na straně klienta, kterým se klient připojuje na server:

```
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();
    }
}
```

Funguje tento páár klient/server dobře? Jak můžeme formálně popsat jeho funkci? Zde je test, který ověřuje, zda je tento výkon „přijatelný“:

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
    Thread[] threads = createThreads();
    startAllThreads(threads);
    waitForAllThreadsToFinish(threads);
}
```

Nastavení je ponecháno tak, aby příklad zůstal jednoduchý (viz „*ClientTest.java*“ na straně 347). Tento test ověřuje, zda se operace dokončí během 10 000 milisekund.

Je to klasický příklad ověřování propustnosti systému. Tento systém by měl zpracovat sérii požadavků klienta během deseti sekund. Pokud bude server schopen zpracovat všechny požadavky klienta včas, testy projdou.

Co se stane, když testy neprojdou? Protože pro výběr zpracování událostí není možné vytvořit nějaký cyklus, nelze toho pro zvýšení rychlosti kódu udělat příliš mnoho. Vyřeší problém použití více podprocesů? Možná, že ano, ale potřebovali bychom vědět, které operace jsou časově náročné. Máme dvě možnosti:

- ◆ Vstupní a výstupní operace – používání soketu, připojování k databázi, čekání na přepínání virtuální paměti atd.
- ◆ Procesor – numerické výpočty, zpracování frekventovaných výrazů, uvolňování paměti atd.

V typických případech mají systémy od každého něco, ale pro danou operaci obvykle jedna z příčin prevládá. Je-li kód závislý na procesoru, lepší hardware může propustnost zvýšit a testy by mohly projít. K dispozici však máme jen určitý počet cyklů centrální procesorové jednotky a přidávání dalších podprocesů problém kódu, vázaného na procesor, nic nevyřeší a operace nebude rychlejší.

Na druhé straně je-li proces závislý na vstupních a výstupních operacích, může uplatnění více podprocesů výkonnost zvýšit. Když jedna část systému čeká na vstupní nebo výstupní operaci, jiná část může této čekací doby využívat, aby zpracovala něco jiného a efektivněji využívala volného strojového času.

## Přidání podprocesů

Předpokládejme na chvíli, že výkonné testy selžou. Jak můžeme zlepšit propustnost tak, aby testy prošly? Je-li metoda `process` na straně serveru závislá na vstupu a výstupu, existuje jeden způsob, jak by mohl server používat podprocesy (stačí jen změnit `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();
}
```

Předpokládejme, že tato změna způsobí, že testy projdou<sup>1</sup>; pak je kód kompletní, že?

1. To si můžete ověřit sami, když si vyzkoušíte starý i nový kód. Podívejte se ještě jednou na kód bez podprocesů na straně 345 a na kód s podprocesy na straně 348.

## Sledování serveru

Aktualizovaný kód na serveru dokončí test úspěšně v čase těsně nad jednu sekundu. Tohle řešení je bohužel poněkud prostoduché a zavádí některé další problémy.

Kolik podprocesů může server vytvořit? Tomu kód neklade žádná omezení, takže bychom mohli dosáhnout limitu, který je dán virtuálním strojem. U mnoha jednoduchých systémů to může stačit. Ale co když má tento systém obsluhovat větší množství uživatelů na veřejné síti? Když se najednou připojí příliš mnoho uživatelů, systém může začít mlít z posledního a zastavit se.

Ponechme ale tento funkční problém prozatím stranou. U uvedeného řešení je problém s čistotou a strukturou. Kolik odpovědností má kód na straně serveru?

- ◆ Správu soketového připojení.
- ◆ Zpracování požadavků klienta.
- ◆ Zásady práce s podprocesy.
- ◆ Zásady vypínání serveru.

Všechny tyto odpovědnosti jsou součástí funkce `process`. Navíc tento kód jde napříč mnoha různými úrovněmi abstrakce. Takže jakkoliv je tato zpracovávající funkce malá, potřebuje předělat.

Existuje několik důvodů, proč by se měl kód na straně serveru předělat. Což znamená, že kód poruší pravidlo jedné odpovědnosti. Abychom měli souběžné systémy čisté, správa podprocesů by měla být soustředěna na místo, odkud je možné tyto podprocesy dobře řídit. Navíc, jakýkoliv kód pro správu podprocesů by neměl dělat nic jiného, než právě spravovat podprocesy. Proč? Když už pro nic jiného, tak proto, že sledování souběžných procesů je natolik komplikované, že si nemůžeme dovolit řešit zároveň žádné jiné problémy.

Jestliže vytvoříme samostatnou třídu pro každou z výše uvedených odpovědností, včetně odpovědnosti za správu podprocesů, pak v případě změny strategie řízení podprocesů bude mít tato změna menší dopad na celkový kód a nezaneřádí ostatní odpovědnosti. To také usnadňuje testování všech ostatních odpovědností, aniž byste se museli starat o podprocesy. Zde je aktualizovaná verze, která to provádí:

```
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();  
}
```

Nyní se všechny otázky, týkající se podprocesů, soustřeďují na jediné místo, kterým je `clientScheduler`. Když nastanou nějaké problémy se souběžnými procesy, máme jediné místo, kde hledat:

```
public interface ClientScheduler {  
    void schedule(ClientRequestProcessor requestProcessor);  
}
```

Implementace zásad souběžnosti je jednoduchá:

```
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();  
    }  
}
```

Když jsme veškerý kód pro řízení podprocesů soustředili na jediné místo, je mnohem jednoduší měnit způsob, jakým podprocesy řídíme. Například šablona Java 5 Executor známená napsat novou třídu a připojit ji k projektu (výpis A.1).

## Závěr

Zavádění souběžnosti do konkrétního příkladu ukazuje možnost, jak zlepšit propustnost systému a jak ověřit tuto propustnost pomocí testovací šablony. Soustředění veškerého kódu, týkajícího se souběžnosti, do malého počtu tříd je příkladem aplikace principu jediné odpovědnosti. V případě souběžného programování je to díky své složitosti zvláště důležité.

## Počet cest při provádění kódu

Podívejte se znova na metodu `incrementValue`, což je jednořádková metoda v jazyce Java, která nemá cykly ani větvení:

```
public class IdGenerator {  
    int lastIdUsed;  
    public int incrementValue() {  
        return ++lastIdUsed;  
    }  
}
```

Možnost přetečení proměnné typu `integer` ignorujte a předpokládejte, že k jediné instanci třídy `IdGenerator` má přístup jen jeden podproces. V tomto případě existuje jen jediná cesta, kterou kód projde, a jediný zaručený výsledek:

- ◆ Návratová hodnota se rovná hodnotě proměnné `lastIdUsed`. Obě budou o jednotku větší, než byly před voláním metody.

### Výpis A.1. 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);
    }
}

```

Co se stane, když ponecháme metodu `beze změn` a použijeme dva podprocesy? Jaké můžeme získat výsledky, když každý z podprocesů jednou zavolá metodu `incrementValue`? Kolik je zde možných cest pro provedení kódu? Zde jsou výstupy (za předpokladu, že počáteční hodnota proměnné `lastIdUsed` je 93):

- ◆ Podproces 1 bude mít hodnotu 94, podproces 2 hodnotu 95 a proměnná `lastIdUsed` je nyní 95.
- ◆ Podproces 1 bude mít hodnotu 95, podproces 2 hodnotu 94 a proměnná `lastIdUsed` je nyní 95.
- ◆ Podproces 1 bude mít hodnotu 94, podproces 2 hodnotu 94 a proměnná `lastIdUsed` je nyní 94.

Poslední výsledek, i když je překvapující, je možný. Abychom viděli, že tyto výsledky jsou reálné, musíme porozumět všem možným cestám, kterými se bude ubírat provádění kódu, a jak jej prostředí JVM provádí.

## Počet cest

Abychom spočetli všechny možné cesty provádění kódu, začneme s vygenerovaným bajtovým kódem. Jeden řádek kódu (`return ++lastIdUsed;`) se přeloží jako osm bajtových instrukcí. Instrukce těchto dvou podprocesů se mohou prokládat tak, jako když karetní hráč míchá balíček karet<sup>2</sup>. I když máme v každé ruce jen osm karet, počet možných kombinací je značný.

V tomto jednoduchém případě máme posloupnost  $N$  instrukcí bez cyklů a podmíněných příkazů a  $T$  podprocesů. Celkový počet proveditelných cest je roven

$$(N \times T)!$$

$$N^T$$

### Výpočet možných řazení

Tohle je obsah e-mailu, který poslal strýček Bob Brettovi:

Máme-li  $N$  kroků a  $T$  podprocesů, máme celkem  $T \times N$  kroků. Před každým krokem je kontextový přepínač, který vybírá mezi  $T$  podprocesy. Každá cesta může být takto reprezentována jako řetězec

2. To je určité zjednodušení, ale za účelem této diskuse můžeme takové zjednodušení použít.

číslic, které vyjadřují kontextové přepínače. Jsou dány kroky A a B a podprocesy 1 a 2. Pak existuje šest možných cest: 1122, 1212, 1221, 2112, 2121, a 2211. Nebo z hlediska kroků je to A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1, a A2B2A1B1. Pro tři vlákna je tato posloupnost 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123 ...

Jednou z charakteristik těchto řetězců je, že vždy musí být  $N$  instancí každého vlákna  $T$ . Takže řetězec 111111 není validní, protože má šest instancí 1 a žádnou instanci 2 nebo 3. Takže chceme permutace jedné z  $N$ , dvou z  $N$  až  $T$  z  $N$ . To jsou permutace  $N \times T$  prvků z  $N \times T$ , což dělá  $(N \times T)!$ , ale bez opakování. Takže vtip spočívá v tom, že spočítáme počet opakování a odečteme je od  $(N \times T)!$ .

Máme-li dva kroky a dva podprocesy, kolik máme opakování? Každý řetězec, skládající se ze čtyř číslic, má dvě jedničky a dvě dvojky. Každý z těchto párů lze prohodit, aniž by se smysl řetězce změnil. Můžete prohodit jedničky nebo obě dvojky, nebo nic. Takže pro každý řetězec existují čtyři isomorfismy, což znamená, že máme tři opakování. Takže tři ze čtyř možností se opakují a stejně tak jedna ze čtyř permutací SE NEOPAKUJE.  $4! \times 0,25 = 6$ . Takže argumentace je zřejmě správná.

Kolik máme opakování? Pro hodnoty  $N=2$  a  $T=2$  se mohu zaměnit jedničky, dvojky nebo obojí. Pro hodnoty  $N=2$  a  $T=3$  se mohu zaměnit jedničky, dvojky, trojky, jedničky a dvojky, jedničky a trojky nebo dvojky a trojky. Záměna prvků znamená tvořit permutace z  $N$ . Řekněme, že máme  $P$  permutací z  $N$ . Počet způsobů, jak tyto permutace poskládat, je  $P^{**}N$ .

Takže počet možných isomorfismů je  $N^T$ . A počet cest je  $(T \times N)! / (N^T)$ . V případě, že  $T=2$  a  $N=2$ , dostáváme 6 (24/4).

Pro  $N=2$  a  $T=3$  dostáváme  $720/8=90$ .

Pro  $N=3$  a  $T=3$  dostáváme  $9!/6^3=1680$ .

Pro jednoduchý případ jednoho řádku v Javě, což se rovná osmi řádkům bajtového kódu a dvěma podprocesům, je celkový počet možných cest pro provedení programu 12 870. Je-li proměnná `lastIdUsed` deklarovaná jako `long`, pak se každé čtení či zápis skládá ze dvou operací namísto jedné a počet možných kombinací je 2 704 156.

Co se stane, když provedeme v této metodě jednu změnu?

```
public synchronized void incrementValue() {
    ++lastIdUsed;
}
```

V obecném případě se při existenci dvou podprocesů změní počet možných cest pro provádění kódu na  $N!$ .

## Hlubší pohled

Jak máme rozumět překvapujícímu závěru, že obě vlákna mohou volat jednou metodu (před přidáním parametru `synchronized`) a obdržíme stejné číselné výsledky? Jak je to možné? Nejdříve to nejdůležitější.

Co je to atomická operace? Atomickou operaci můžeme definovat jako libovolnou operaci, kterou nelze přerušit. Například v následujícím kódu instrukce na řádku 5, kde se do proměnné `lastId` vkládá nula, je atomická, protože podle pravidel správy paměti Java Memory Model (JMM) je vkládání hodnoty do 32bitové proměnné nepřerušitelné.

```

01: public class Example {
02:     int lastId;
03:
04:     public void resetId() {
05:         value = 0;
06:     }
07:
08:     public int getNextId() {
09:         ++value;
10:    }
11: }

```

Cose stane, když změníme typ proměnné `lastId` z `int` na `long`? Bude řádek 5 atomický? Podle specifikací JVM nikoliv. Operace by na nějakém konkrétním procesoru mohla být atomická, ale podle specifikací JVM vyžaduje přiřazení 64bitové hodnoty dvě přiřazení po 32 bitech. To znamená, že mezi prvním 32bitovým přiřazením a druhým by se mohl nějaký jiný podproces vtrít do zpracování a některou z hodnot změnit.

Co můžeme říci o operátoru inkrementace předem, `++` na řádku 9? Tento operátor lze přerušit a ne-ní tudíž atomický. Abychom tomu dobré rozuměli, podívejme se na bajtový kód obou těchto metod podrobněji.

Než půjdeme dále, zde jsou tři definice, které pro nás budou v dalším textu důležité.

- ◆ Rámec – Každé volání metody vyžaduje rámec (zásobníku). Rámec obsahuje návratovou adresu, parametry, které se předávají metodě, a lokální proměnné, jež jsou v metodě definované. Tohle je standardní technika, která v moderních jazycích určuje obsah zásobníku, aby bylo proveditelné základní volání funkce či metody a aby bylo možné jejich rekursivní volání.
- ◆ Lokální proměnná – Jakákoliv proměnná, definovaná v rozsahu metody. Všechny nestatické metody mají minimálně jednu proměnnou, `this`, která reprezentuje aktuální objekt, ten, jenž obdržel poslední zprávu (v aktuálním podprosesu) a který způsobil volání metody.
- ◆ Zásobník operandů – Mnoho instrukcí JVM má parametry. Ty se ukládají do zásobníku operandů. Je to standardní datová struktura typu LIFO (last-in, first-out)

Bajtový kód, který je vygenerovaný pro `resetId()`:

Instrukce	Popis	Stav zásobníku po operaci
ALOAD_0	Vložte nultou proměnnou do zásobníku operandů. Která proměnná je nultá? Je to <code>this</code> , odkaz na aktuální objekt. V okamžiku volání metody se příjemce této zprávy, instance třídy <code>Example</code> , vloží do pole proměnných rámce, který byl vytvořen voláním metody. Je to vždy první proměnná, která se vkládá do rámce každé instance metody.	<code>this</code>
ICONST_0	Vložte konstantní hodnotu 0 do zásobníku operandů. <code>this</code> .	0
PUTFIELD lastId	Uložte hodnotu z vrcholu zásobníku (což je nula) do složky objektu, na který vede odkaz z objektu <code>this</code> , který je pod vrcholem zásobníku.	<empty>

Tyto tři instrukce jsou zcela určitě atomické, protože ačkoliv podproces, který je vykonává, může být po každé z těchto instrukcí kdykoliv přerušen, informace pro instrukci PUTFIELD (konstantní hodnota 0 na vrcholu zásobníku a odkaz na `this` pod ním spolu s datovou složkou) nemůže být ovlivněna jiným vlákнем. Takže když dojde k přiřazení, máme jistotu, že hodnota nula se uloží do pole hodnot. Operace je atomická. Všechny operandy se týkají informace, která je vůči metodě lokální, takže jednotlivé podprocesy se nemohou navzájem rušit.

Takže pokud se tyto tři instrukce provedou v deseti podprocesech, existuje  $4,38679733629e+24$  možných kombinací. Avšak existuje pouze jeden možný výstup, takže různá řazení nejsou relevantní. Prostě platí, že stejný výstup je v tomto případě zajištěn i pro typ long. Proč? Všech deset podprocesů přiřazuje konstantní hodnotu. Přestože je provádění jejich instrukcí prokládané, konečný výsledek je stejný.

S operací přiřízení jedné předem ++ budou problémy. Předpokládejme, že na začátku metody má proměnná `lastId` hodnotu 42. Bajtový kód nové metody je zde:

Instrukce	Popis	Stav zásobníku po operaci
ALOAD 0	Vlož objekt <code>this</code> do zásobníku operandů.	<code>this</code>
DUP	Zkopíruj nejvyšší položku zásobníku. Nyní máme v zásobníku dvě kopie objektu <code>this</code> .	<code>this, this</code>
GETFIELD <code>lastId</code>	Vyhledejte hodnotu složky <code>lastId</code> z vrcholu zásobníku ( <code>this</code> ) a uložte tuto hodnotu zpátky do zásobníku.	<code>this, 42</code>
ICONST_1	Vložte do zásobníku celočíselnou konstantu 1.	<code>this, 42, 1</code>
IADD	Celočíselně sečtěte na zásobníku dvě nejvyšší hodnoty a výsledek uložte zpět do zásobníku.	<code>this, 43</code>
DUP_X1	Okopírujte hodnotu 43 a uložte ji před objekt <code>this</code> .	<code>43, this, 43</code>
PUTFIELD <code>value</code>	Uložte hodnotu 43 na vrcholu zásobníku do složky aktuálního objektu, který je reprezentován hodnotou bezprostředně pod vrcholem zásobníku ( <code>this</code> ).	<code>43</code>
IRETURN	Vraťte hodnotu na vrcholu zásobníku (a jedinou v něm). <empty>	

Představte si případ, kdy první podproces ukončil první tři instrukce včetně operace GETFIELD a poté byl přerušen. Druhý podproces převzal řízení, provedl celou metodu, zvýšil proměnnou `lastId` o jedničku a obdržel číslo 43. Poté první podproces pokračoval tam, kde přestal. Hodnota 42 je stále v zásobníku, protože to byla hodnota proměnné `lastId` během provádění operace GETFIELD. K ní přidá jedničku, získá hodnotu 43 a výsledek uloží. Návratová hodnota prvního podprocesu je rovněž 43. Výsledkem je, že jeden z inkrementů je ztracen, protože první podproces vystřídal druhý, poté co druhý přerušil práci prvního. Problém vyřešíme tím, že metodu `getNextId()` synchronizujeme.

## Závěr

Abychom porozuměli tomu, jak může jeden podproces vstoupit do činnosti druhého, nemusíme detailně rozumět bajtovému kódu. Jestliže porozumíte příkladu, který měl ukázat, jak může jeden podproces přerušit druhý, bude to stačit.

Tento triviální příklad ukazuje, že je třeba dostatečně znát paměťový model, abyste věděli, co je a co není bezpečné. V této věci panuje obecně chybný názor, že inkrementace o jedničku, ať už před nebo po operaci, je atomická. Rozhodně není. To znamená, že potřebujete znát:

- ◆ Kde se nacházejí sdílené objekty nebo hodnoty.
- ◆ kód, který může způsobit problémy souběžného čtení nebo aktualizace,
- ◆ jak takové souběžné záležitosti hlídat, aby nenastaly.

## Vyznejte se ve své knihovně

### Běhový rámec

Jak jsme si již ukázali v případě `ExecutorClientScheduler.java` na straně 326, běhový rámec, zavedený v Javě 5, umožňuje rafinované provedení kódu za použití fondu podprocesů. Je to třída v balíčku `java.util.concurrent`.

Pokud vytváříte podprocesy a nepoužíváte fond podprocesů nebo *používáte* nějaký ručně psaný, měli byste uvažovat o použití běhového rámce. Váš kód bude čistší, srozumitelnější a kratší.

Běhový rámec vytvoří fond podprocesů, automaticky změní jejich velikost a obnoví je, je-li to zapotřebí. Podporuje také konstrukce zvané *future*, což je obecná konstrukce pro souběžné programování. Běhový rámec pracuje s třídami, které implementují rozhraní `Runnable` nebo `Callable`. První rozhraní se podobá druhému s tím, že rozhraní `Callable` může vracet výsledek, což je u úloh s více podprocesy všeobecně zapotřebí.

Konstrukce *future* je užitečná, když je třeba provést souběžné a nezávislé operace a když musíme čekat na jejich ukončení.

```
public String processRequest(String message) throws Exception {  
    Callable<String> makeExternalCall = new Callable<String>() {  
        public String call() throws Exception {  
            String result = "";  
            // make external request  
            return result;  
        }  
    };  
    Future<String> result = executorService.submit(makeExternalCall);  
    String partialResult = doSomeLocalProcessing();  
    return result.get() + partialResult;  
}
```

V tomto příkladě spustí metoda objekt `makeExternalCall`. Pak pokračuje v dalším zpracování. Poslední řádek volá metodu `result.get()`, která další činnost zablokuje, dokud neskončí *future*.

### Řešení bez blokace

VM jazyka Java 5 má výhodu moderního návrhu procesoru, který podporuje aktualizace dat spolehlivě a bez blokace. Podívejte se například na třídu, která používá synchronizaci (a tudíž blokaci), aby zabezpečila bezpečnou aktualizaci hodnoty při použití více podprocesů:

```
public class ObjectWithValue {  
    private int value;  
    public void synchronized incrementValue() { ++value; }  
    public int getValue() { return value; }  
}
```

Java 5 má několik nových tříd pro podobné situace: `AtomicBoolean`, `AtomicInteger` a `AtomicReference` jsou tři příklady. Je jich tam několik navíc. Výše uvedený kód můžeme napsat jinak a vyhnout se blokování:

```
public class ObjectWithValue {  
    private AtomicInteger value = new AtomicInteger(0);  
    public void incrementValue() {  
        value.incrementAndGet();  
    }  
    public int getValue() {  
        return value.get();  
    }  
}
```

I když tento kód používá objekt namísto primitivního typu a posílá zprávu `incrementAndGet()` místo inkrementování, výkon této třídy bude téměř vždy lepší, než je u předchozí verze. V některých případech bude jen o něco rychlejší, ale téměř nikdy nebude pomalejší.

Jak je to možné? Moderní procesory mají jednu operaci s typickým názvem `Compare and Swap` (CAS – porovnej a zaměň). Tato operace je u databází analogická optimistickému zamykání, zatímco synchronizující verze je analogická pesimistickému zamykání.

Klíčové slovo `synchronized` znamená vždy zamykání, i když se druhý podproces nesnaží tutéž hodnotu aktualizovat. I když se v dalších verzích výkon při použití vnitřních zámků zlepšuje, je to stále nákladné.

Verze bez blokování začíná s předpokladem, že podprocesy v běžných případech nemodifikují stejné hodnoty tak často, aby z toho vznikl problém. Místo toho efektivně detekují, zda taková situace nastala, a pokoušeji se operaci opakovat, dokud se aktualizace nepodaří. Tato detekce je téměř vždy efektivnější než použití zámků, a to i za běžných obyčejných podmínek, i za podmínek silného nedostatku zdrojů.

Jak to JVM dokáže? Operace CAS je atomická. Je logické, že operace CAS vypadá takto:

```
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;  
}
```

Když se metoda pokouší aktualizovat sdílenou proměnnou, ověřuje si operace CAS, zda má nastavovaná proměnná stále poslední známou hodnotu. Jestliže ano, pak se tato proměnná aktualizuje. Když ne, proměnná se neaktualizuje, protože tomu stojí v cestě jiný podproces. Metoda, která se o to pokouší (za použití operace CAS), vidí, že změna se nepodařila, a pokusí se o to znova.

## Bezpečné třídy bez podprocesů

Existuje několik tříd, které nejsou ze své podstaty vzhledem k podprocesům bezpečné. Zde je několik příkladů:

- ◆ `SimpleDateFormat`
- ◆ `Databázové spojení`
- ◆ `Kontejnery v java.util`
- ◆ `Servlety`

Všimněte si, že některé třídy kolekcí mají jednotlivé metody, které jsou z hlediska podprocesů bezpečné. Avšak jakákoli operace, která v sobě zahrnuje volání více než jedné metody, bezpečná není. Pokud například nechcete něco nahradit v tabulce typu `HashTable`, protože to tam už je, můžete napsat následující kód:

```
if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

Každá jednotlivá metoda je z hlediska podprocesů bezpečná. Avšak jiný podproces by mohl přidat hodnotu mezi voláním metody `containsKey` a `put`. Pro řešení tohoto problému existuje několik možností.

- ◆ Zamkněte nejdříve `HashTable` a zajistěte, ať všichni ostatní uživatelé tohoto objektu udělají totéž – zamykání u klienta:

```
synchronized(map) {
    if(!map.containsKey(key))
        map.put(key,value);
}
```

- ◆ Zabalte `HashTable` do svého vlastního objektu a použijte něco jiného z API – zamykání na serveru za použití návrhového vzoru ADAPTÉR:

```
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);
    }
}
```

- ◆ Použijte bezpečné kolekce:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();
map.putIfAbsent(key, value);
```

Kolekce v balíčku `java.util.concurrent` obsahují operace jako `putIfAbsent()`, které tyto operace umožňují.

## Závislost mezi metodami může souběžný kód porušit

Zde máme triviální příklad, jak zavést závislosti mezi metodami:

```
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;
    }
}
```

Zde je kód, který používá třídu `IntegerIterator`:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    // do something with nextValue
}
```

Když bude tento kód prováděn pouze jedním podprocesem, nevznikne žádný problém. Ale co se stane, když se dva podprocesy pokusí sdílet jednu instanci třídy `IntegerIterator` se záměrem, aby každý podproces zpracoval hodnoty, které obdrží, ale že každý prvek seznamu bude zpracován pouze jednou? Většinou se nestane nic. Podprocesy budou sdílet seznam bez problémů, zpracují prvky dané instancí `iterator`, a skončí, když iterátor doběhne na konec. Existuje však malá pravděpodobnost, že na konci iterace si dva podprocesy budou navzájem překážet a způsobí, že jeden podproces se dostane za konec procházeného úseku a vyvolá výjimku.

Problém je v tomto: podproces 1 vyšle dotaz `hasNext()`, jehož návratová hodnota je `true`. Pak je podproces 1 nuceně přerušen a podproces 2 vyšle týž dotaz, jehož návratová hodnota je stále `true`. Podproces 2 pak zavolá metodu `next()`, která vrátí očekávanou hodnotu, ale má vedlejší efekt, když způsobí, že návratová hodnota metody `hasNext()` bude `false`. Podproces 1 začne znova a předpokládá, že metoda `hasNext()` stále vrací `true`, a zavolá metodu `next()`. I když jsou samostatné metody synchronizované, klient používá metody dvě.

Tohle je skutečný problém a příklad ze třídy problémů, které vznikají při práci se souběžným kódem. V této konkrétní situaci je problém zvlášť nezřetelný, protože jediný okamžik, kdy dochází k chybě, nastává během závěrečného kroku iterátoru. Jestliže podprocesy selžou v ten pravý okamžik, pak by

jeden z nich mohl překročit konec iterovaného rozsahu. Je to druh chyby, ke kterému dochází dlouho poté, co byl systém dán do ostrého provozu, a je velmi těžké ji vystopovat.

Máte tři možnosti:

- ◆ Tolerovat selhání.
- ◆ Vyřešit problém změnou klienta: zavést zamykání na straně klienta.
- ◆ Vyřešit problém změnou na straně serveru, což následně povede ke změnám u klienta: zamykání na straně serveru.

## Tolerovat selhání

Někdy můžete vše zařídit tak, aby selhání systému nezpůsobilo žádné škody. Uvedený klient by například mohl zachytit výjimku a ošetřit ji. Upřímně řečeno, je to poněkud nedbalé. Je to jako odstraňení bloků ztracené paměti tím, že o půlnoci restartujeme systém.

## Zamykání na straně klienta

Aby třída `IntegerIterator` pracovala s více podprocesy korektně, změňte kód klienta (a všechny ostatní) takto:

```
IntegerIterator iterator = new IntegerIterator();
while (true) {
    int nextValue;
    synchronized (iterator) {
        if (!iterator.hasNext())
            break;
        nextValue = iterator.next();
    }
    doSomethingWith(nextValue);
}
```

Každý klient provede uzamknutí pomocí klíčového slova `synchronized`. Tohle zdvojení porušuje princip neopakování (Don't repeat yourself principle, DRY), ale možná, že je to nezbytné, pokud kód používá nástroje třetí strany, které nejsou s ohledem na podprocesy bezpečné.

Tato strategie je riskantní, protože všichni programátoři, kteří používají server, si musí pamatovat, že je nutné zdroj před operací uzamknout a po jejím ukončení uvolnit. Před mnoha (mnoha!) lety jsem pracoval na systému, který používal zamykání sdíleného zdroje na straně klienta. Zdroj se používal na stovkách různých míst po celém kódu. Na jednom z těchto míst zapomněl nějaký špatný programátor zdroj uzamknout.

Systém pracoval v režimu sdílení času a na mnoha terminálech zpracovával účetní agendu odborového svazu autodopravců Local 705. Počítac byl nainstalován na zvýšené podlaze v místnosti s klimatizací 50 mil severně od ústředí organizace. Měli tam desítky úředníků, zadávajících příspěvky členů do terminálů. Ty byly spojeny s počítacem pomocí dedikované telefonní linky s poloduplexními modemy při rychlosti 600 bps. (Je to velmi, *velmi* dávno.)

Přibližně jednou denně se jeden z terminálů „kousl“. Nebylo v tom žádné pravidlo nebo důvod. Zablokování si mezi terminály nevybíralo a docházelo k němu kdykoliv. Jako by někdo házel kostkou a vybíral čas a terminál, který se má zablokovat. Někdy se zablokovalo více terminálů. Někdy naopak uplynuly dny a nic se nedělo.

Zpočátku byl jediným řešením restart. Ale restarty bylo obtížné koordinovat. Museli jsme zavolat na ústředí a umožnit všem, aby na všech terminálech ukončili rozdělanou práci. Poté jsme museli vypnout systém a restartovat jej. Pokud někdo pracoval na něčem důležitém, trvalo to hodinu nebo dvě. Zablokovaný terminál musel prostě zůstat zablokovaný.

Po několika týdnech ladění jsme zjistili, že příčinou byl čítač cyklické vyrovnávací paměti, který vypadl se svým ukazatelem ze synchronizace. Tato vyrovnávací paměť řídila výstup z terminálu. Hodnota ukazatele indikovala, že paměť byla prázdná, ale čítač říkal, že byla plná. S prázdnou pamětí nebylo co zobrazovat, ale protože byla zároveň plná, nikdo do ní nemohl nic zapsat a nic se nemohlo na obrazovce ukázat.

Věděli jsme tedy, proč se terminály blokovaly, ale nevěděli jsme, proč nebyla cyklická vyrovnávací paměť synchronizovaná. Pořídili jsme si tedy nějaké programové pomůcky, abychom mohli na problém pracovat. Něco bylo možné vyčíst z přepínačů na předním panelu počítače. (Bylo to velmi, velmi, *velmi* dávno.) Napsali jsme si krátkou programovou past, která uměla detektovat okamžik, kdy byl jeden z těchto stavů vyvolán, a pak jsme se podívali na cyklickou vyrovnávací paměť, jež byla plná i prázdná zároveň. Když takový stav nastal, byla vyrovnávací paměť resetována a byla prázdná. A hle! Zablokovaný terminál začal opět zobrazovat data.

Při zablokování terminálu jsme tedy nemuseli restartovat systém. Lidé z vedení si nás prostě zavolali a řekli nám, že nastalo zablokování, a my jsme přišli do místonosti s počítačem a zmáčkli jsme příslušný přepínač.

Samozřejmě že lidé z vedení pracovali někdy o víkendech, kdežto my nikoliv. Tak jsme přidali do plánovače funkci, která jednou za minutu kontrolovala všechny vyrovnávací paměti. Tato funkce je resetovala, pokud u kterékoliv z nich nastal stav, že paměť byla plná i prázdná. To uvolnilo terminály dříve, než by se lidé z vedení vůbec dostali k telefonu.

Po několika dalších týdnech podrobného studia všech stránek fádního assemblerového kódu jsme našli viníka. Provedli jsme si propočty a zjistili jsme, že četnost zablokování souvisela s jedním nechráněným použitím cyklické vyrovnávací paměti. Tohle místo jsme museli nalézt. Bohužel to bylo tak dávno, že jsme neměli žádné nástroje na prohledávání nebo na křížové reference, ani žádný jiný druh automatické pomoci. Museli jsme si prostě podrobně prostudovat výpisy.

Tehdy, v Chicagu za chladné zimy roku 1971, jsem dostal důležitou lekci. Zamykání na straně klienta může být opravdu zničující.

## Zamykání na straně serveru

Zdvojení můžeme odstranit pomocí následujících změn ve třídě `IntegerIterator`:

```
public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;
    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
}
```

A kód na straně klienta změníme také:

```
while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // do something with nextValue
}
```

V tomto případě jsme u naší třídy v podstatě změnili API, abychom mohli pracovat s více podprocesy<sup>3</sup>. Klient musí místo volání `hasNext()` provést kontrolu na hodnotu `null`.

Obecně byste měli dát přednost zamykání na straně serveru z následujících důvodů:

- ◆ Omezuje opakování kódu – zamykání na straně klienta nutí každého klienta zamyskat správně server. Když tento kód umístíme na server, klient již nemusí tento objekt používat a nemusí se starat o další kód, který by uzamknutí provedl.
- ◆ Umožňuje lepší výkon – v případě použití softwaru bez podprocesů můžete vyměnit bezpečný server pro podprocesy za server, který bezpečný není, čímž se vyhnete všem režijním nákladům.
- ◆ Snižuje pravděpodobnost chyb – stačí jen nepozornost jednoho programátora, který nenastaví ten správný zámek.
- ◆ Vede k jedné zásadě – zásady jsou na jednom místě – na serveru a ne na více místech, jako třeba na všech klientech.
- ◆ Omezuje rozsah sdílených proměnných – klient neví ani o proměnných, ani o tom, jak se zamýkají. Vše je ukryto na serveru. Když něco selže, počet možných míst, kde došlo k chybě, je menší.

Co když kód na serveru nevlastníte?

- ◆ Použijte vzor ADAPTÉR, změňte aplikační rozhraní a přidejte kód pro zamykání:

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();
    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- ◆ Nebo ještě lépe, použijte bezpečné kolekce s rozšířeným rozhraním.

## Zvyšování propustnosti

Předpokládejme, že chceme putovat po síti a čist obsah nějakých stránek ze seznamu URL. Během čtení budeme stránku analyzovat a shromažďovat nějaká statistická data. Když budou stránky načteny, vytiskneme souhrnnou zprávu.

Následující třída vrací obsah jedné stránky na základě jedné adresy URL.

3. Ve skutečnosti není rozhraní `Iterator` ze své podstaty při práci s podprocesy bezpečné. Nikdo jej nevyvíjel se zítelem k prostředí s více podprocesy, takže by to nemělo být překvapením.

```
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();  
        }  
    }  
}
```

Následující třída je iterátor založený na iterátoru daných adres URL, který vrací obsah :

```
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);  
    }  
}
```

Instanci třídy PageIterator lze sdílet mnoha různými podprocesy, z nichž každý používá svou vlastní instanci třídy PageIterator, aby bylo možné číst a analyzovat stránky získané iterátorem.

Všimněte si, že blok synchronized je velmi krátký. Obsahuje pouze kritickou sekci hluboko uvnitř třídy PageIterator. Vždy je lepší synchronizovat co nejméně kódu a ne naopak.

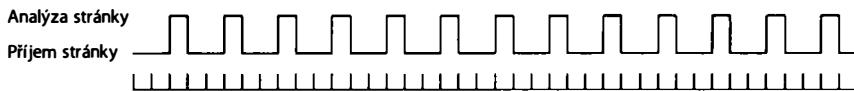
## Kalkulace propustnosti jednoho podprocesu

Proveďme nyní několik jednoduchých výpočtů. Abychom mohli argumentovat, předpokládejme, že platí následující čísla:

- ◆ Průměrný čas na přijetí stránky: 1 sekunda.
- ◆ Průměrná doba zpracování jedné stránky: 0,5 sekundy.
- ◆ Vstupní a výstupní operace nevyžadují žádný strojový čas CPU, zatímco zpracování stránek CPU vytěžuje na sto procent.

Jestliže jeden podproces zpracovává  $N$  stránek, bude celková doba zpracování  $1,5 \times N$  sekund. Obrázek A.1 ukazuje snímek zpracování 13 stránek během přibližně 19,5 sekundy.

#### Jeden podproces



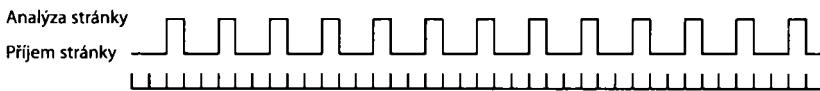
Obrázek A.1. Jeden podproces

## Kalkulace propustnosti při více podprocesech

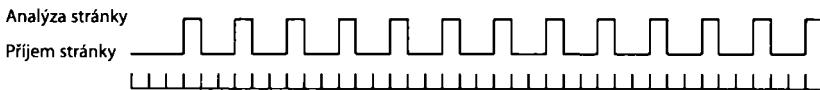
Je-li možné získávat stránky v libovolném pořadí a nezávisle je zpracovávat, pak je možné používat více podprocesů a zvýšit tím propustnost. Co se stane, když použijeme tři podprocesy? Kolik stránek získáme za stejnou dobu?

Jak můžete vidět na obrázku A.2, řešení s více podprocesy umožňuje, aby se analýza stránek překrývala s jejich čtením. V ideálním světě to znamená, že procesor je plně využit. Každá stránka se čte jednu sekundu a tato operace se překrývá se dvěma analyzami. Takto můžeme zpracovávat dvě stránky za sekundu, což je třikrát vyšší propustnost než u řešení s jedním podprocesem.

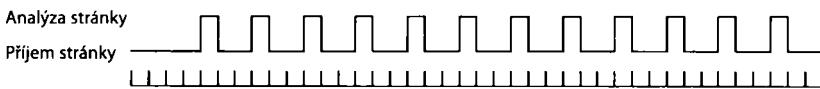
#### Podproces 1



#### Podproces 2



#### Podproces 3



Obrázek A.2. Tři souběžné podprocesy

## Zablokování

Představte si webovou aplikaci se dvěma sdílenými fondy zdrojů, které mají konečnou velikost:

- ◆ Fond databázových připojení pro lokální činnost v úložišti procesů.
- ◆ Fond připojení MQ k hlavnímu úložišti dat.

Předpokládejme, že v této aplikaci jsou dvě operace, „vytvoř“ a „aktualizuj“:

- ◆ „Vytvoř“ – získá připojení k hlavnímu úložišti a pak k databázi. Komunikuje se službami hlavního úložiště a poté uloží lokální práci v databázi procesů.
- ◆ „Aktualizuj“ – získá připojení k databázi a pak k hlavnímu úložišti. Čte z práce uložené v databázi procesů a posílá výsledky do hlavního úložiště.

Co se stane, když bude více uživatelů, než je kapacita fondů? Dejme tomu, že velikost každého fondu je deset.

- ◆ Deset uživatelů se pokusí provést operaci „vytvoř“, takže získají všech deset databázových připojení. Každý podproces bude přerušen poté, co obdrží databázové připojení, ale před zřízením připojení k hlavnímu úložišti.
- ◆ Deset uživatelů se pokusí provést operaci „aktualizuj“, takže získají všech deset připojení k hlavnímu úložišti. Každý podproces bude přerušen poté, co získá připojení k hlavnímu úložišti, ale před získáním připojení k databázi.
- ◆ Nyní deset podprocesů, provádějících operaci „vytvoř“, musí čekat, dokud neobdrží připojení k hlavnímu úložišti, ale deset podprocesů provádějících „aktualizaci“ musí čekat, dokud nezíská připojení k databázi.
- ◆ Zablokování. Systém se nikdy nevzpamatuje.

To může vypadat nepravděpodobně, ale kdo by si přál systém, který totálně zamrzne každý druhý týden? Kdo by chtěl ladit systém se symptomy, které se dají tak obtížně reprodukovat? Je to druh problémů, ke kterým v praxi dochází a jejichž řešení zabírá týdny.

Typickým „řešením“ je vložit ladící příkazy, které mají pomoci při zjišťování, k čemu dochází. Samozřejmě že ladící příkazy změní kód natolik, že k vzájemným zablokováním začne docházet v jiných situacích, a trvá měsíce, než k nim dojde znova<sup>4</sup>. Chceme-li opravdu problém zablokování vyřešit, musíme porozumět jeho příčinám. Existují tři podmínky, které jsou zapotřebí k tomu, aby došlo k zablokování:

- ◆ Vzájemné vyloučení.
- ◆ Zamkní a čkej.
- ◆ Zdroj nelze získat nucenou výměnou.
- ◆ Cyklické čekání.

## Vzájemné vyloučení

Vzájemné vyloučení nastává, když několik podprocesů potřebuje používat tytéž zdroje, pro něž platí, že:

- ◆ Více podprocesů je nesmí používat souběžně.
- ◆ Je jich omezený počet.

Obecným příkladem takového zdroje je databázové připojení, soubor, který je otevřený pro zápis, zámek na datovém záznamu nebo semafor.

4. Například někdo vloží příkaz pro ladící výstup a problém „zmizí“. Ladící kód „opraví“ problém, který ale dále zůstává v systému.

## Zamkní a čekej

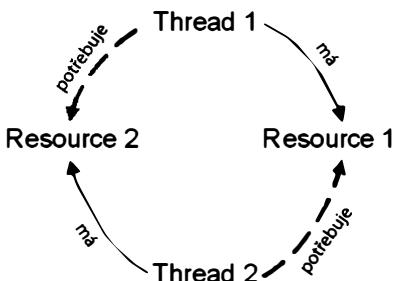
Když podproces konečně získá zdroj, neuvolní jej, dokud nezíská všechny ostatní zdroje, které potřebuje, a nedokončí svou práci.

## Zdroj nelze získat nucenou výměnou

Jeden podproces nemůže převzít zdroje jiného podprocesu. Má-li jednou nějaký podproces zdroj, jedinou možností, jak by jej jiný podproces mohl získat, je, že jej současný majitel uvolní.

## Cyklické čekání

Tomu se také říká smrtelné objetí. Představte si dva podprocesy, T1 a T2, a dva zdroje R1 a R2. T1 vlastní R1 a T2 vlastní R2. T1 ale požaduje také R2 a T2 požaduje R1. To vede k situaci na obrázku A.3:



Obrázek A.3. Cyklické čekání

Aby mohlo dojít k zablokování, musí být splněny všechny čtyři podmínky. Nebude-li splněna kterákoliv z těchto podmínek, není zablokování možné.

## Řešení vzájemného vyloučení

Jednou ze strategií, jak předcházet zablokování, je vyhnout se jeho podmínkám. Toho můžete dosáhnout několika způsoby:

- ◆ Použít zdroje, které lze využívat současně. Například třídu `AtomicInteger`.
- ◆ Zvýšit počet zdrojů tak, aby se rovnal nebo převyšil počet soutěžících podprocesů.
- ◆ Zkontrolovat, zda jsou všechny zdroje volné, ještě než se podproces pokusí nějaký získat.

Bohužel je počet většiny zdrojů omezený a nelze je použít simultánně. A není neobvyklé, že se o druhém zdroji rozhodne až na základě výsledků toho prvního. Ale nebojte se, zbyvají ještě tři možnosti.

## Řešení problému „Zamkní a čekej“

Vzájemné zablokování můžete odstranit také tím, že nebudete čekat. Zkontrolujte každý zdroj před jeho přivlastněním, a pokud narazíte na nějaký, který je již zadán, uvolněte je všechny a začněte znova.

U tohoto přístupu existuje několik potenciálních problémů:

- ◆ Hladovění – jeden podproces není permanentně schopen získat zdroje, které potřebuje (možná, že má jedinečnou kombinaci zdrojů, jež jsou k dispozici jen velmi zřídka).
- ◆ Aktivní zablokování – několik podprocesů se rozbehne a všechny jeden zdroj získají a jeden zdroj uvolní, pořád dokola. To se stává zvláště u jednodušších algoritmů pro plánování činnosti CPU (například u zabudovaných zařízení nebo u jednodušších ručně psaných algoritmů pro vyvážený chod podprocesů).

Oba problémy mohou způsobit špatnou propustnost. První z nich vede k malému využití CPU, druhý naopak k vysokému, ale neúčelnému využití CPU.

I když se tato strategie může zdát neefektivní, je to lepší než nic. Má výhodu v tom, že ji lze použít kdykoliv, když vše ostatní selže.

## Řešení získávání zdrojů nucenou výměnou

Jinou strategií, jak se vyhnout zablokování, je umožnit podprocesům, aby přebíraly zdroje jiným podprocesům. To se obvykle dělá pomocí nějakého jednoduchého mechanismu požadavků. Když podproces zjistí, že je zdroj zaneprázdněný, požádá vlastníka, aby jej uvolnil. Pokud i vlastník čeká na nějaké další zdroje, všechny je uvolní a začíná znovu.

To je podobné předchozímu přístupu, ale má to výhodu, že podproces může na zdroj čekat. To sníží počet opakování startů. Ale varuji vás, protože řízení všech těchto požadavků může být velmi složité.

## Řešení cyklického čekání

Tohle je nejobvyklejší přístup, jak předejít zablokování. U většiny systému nevyžaduje nic více než jednoduchou konvenci schválenou všemi stranami.

Ve výše uvedeném příkladě s Podprocesem 1, který požaduje Zdroj 1 a Zdroj 2 a s Podprocesem 2, jenž také vyžaduje Zdroj 1 a Zdroj 2, stačí, když přinutíme oba podprocesy, aby alokovaly zdroje ve stejném pořadí. Tím se zablokování vlivem cyklického čekání vyloučí.

Obecněji, jestliže budou všechny podprocesy souhlasit s globálním řazením zdrojů, a jestliže zdroje alokují v tomto pořadí, nebude vzájemné zablokování možné. Podobně jako jiné strategie, i tato může způsobit problémy:

- ◆ Pořadí získávání zdrojů nemusí odpovídat pořadí jejich používání. Tak se může stát, že zdroj, získaný na začátku, se použije až na konci. To může způsobit, že zdroje budou blokovány déle, než je nutné.
- ◆ V některých případech není možné zavést pořadí na získávání zdrojů. Jestliže identifikátor druhého zdroje vzniká jako výsledek operace s prvním zdrojem, není možné pořadí stanovit.

Existuje tedy mnoho způsobů, jak se zablokování vyhnout. Některé vedou k hladovění, zatímco jiné příliš zatěžují CPU a zpomalují reakční dobu. TANSTAAFL<sup>5</sup>! Izolování těch částí vašeho řešení

5. Nic se nevyrovnaná jídlo zadarmo. (There ain't no such thing as a free lunch.)

s podprocesy, aby bylo možné jejich ladění a experimentování, je důležitý způsob, jak získat náhled a rozhodnout se pro nejlepší strategie.

## Testování kódu s více podprocesy

Jak můžeme napsat testy, které by ukázaly, že následující kód je chybný?

```
01: public class ClassWithThreadingProblem {  
02:     int nextId;  
03:  
04:     public int takeNextId() {  
05:         return nextId++;  
06:     }  
07: }
```

Zde je popis testu, který je schopen dokázat, že je uvedený kód vadný:

- ◆ Uložit aktuální hodnotu proměnné `nextId`.
- ◆ Vytvořit dva podprocesy a oba nechat, aby jedenkrát zavolaly metodu `takeNextId()`.
- ◆ Ověřit, že proměnná `nextId` je o dvě větší, než na počátku.
- ◆ Nechte tento kód, dokud si neukážeme, že hodnota proměnné `nextId` se zvýšila pouze o jedničku a ne o dvojku.

Takový test je na výpisu A.2.

### Výpis A.2. 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  
11:             = new ClassWithThreadingProblem();  
12:         Runnable runnable = new Runnable() {  
13:             public void run() {  
14:                 classWithThreadingProblem.takeNextId();  
15:             }  
16:         };  
17:         for (int i = 0; i < 50000; ++i) {  
18:             int startingId = classWithThreadingProblem.lastId;  
19:             int expectedResult = 2 + startingId;  
20:             Thread t1 = new Thread(runnable);  
21:             Thread t2 = new Thread(runnable);  
22:             t1.start();  
23:             t2.start();
```

```
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("Měl ukázat problémy s podprocesy, ale neukázal.");
36:   }
37: }
```

Řádek	Popis
10	Vytvořte jedinou instanci třídy <code>ClassWithThreadingProblem</code> . Všimněte si, že musíme použít klíčové slovo <code>final</code> , protože ji používáme níže v anonymní vnitřní třídě.
12–16	Vytvořte anonymní vnitřní třídu, která používá jedinou instanci metody <code>ClassWithThreadingProblem</code> .
18	Tento kód spusťte vícekrát. Dostatečněkrát na to, aby bylo možné ukázat, že kód selhal, ale netolikrát, aby to trvalo příliš dlouho. Je to otázka kompromisu. Nechceme na selhání čekat příliš dlouho. Vybrat takové číslo není jednoduché – ačkoliv později uvidíme, že ho můžeme značně snížit.
19	Zapamatujte si počáteční hodnotu. Tento test se pokouší dokázat, že kód v metodě <code>ClassWithThreadingProblem</code> není správný. Pokud test projde, dokáže tím, že kód není správný. Pokud test selže, pak nebyl schopen dokázat, že je kód špatný.
20	Předpokládáme, že konečná hodnota bude o dva vyšší, než je hodnota současná.
22–23	Vytvořte dva podprocesy. Oba budou používat objekt, který jsme vytvořili na řádcích 12–16. To nám poskytuje potenciál dvou podprocesů, které se pokusí použít jedinou instanci metody <code>ClassWithThreadingProblem</code> a zasahovat do činnosti toho druhého.
24–25	Uveďte naše dva podprocesy do chodu.
26–27	Čekejte, dokud oba podprocesy nedokončí činnost, a pak zkontrolujte výsledky.
29	Uložte aktuální konečnou hodnotu.
31–32	Lišila se proměnná <code>endingId</code> od očekávané hodnoty? Pokud ano, provedte příkaz <code>return</code> a ukončete test – dokázali jsme, že je kód chybný. Pokud ne, zkuste to ještě jednou.
35	Pokud jsme se dostali až sem, náš test nebyl schopný v „rozumném“ čase dokázat, že je ostrý kód chybný. Náš kód tedy selhal. Bud ostrý kód chybný není, nebo jsme jej nenechali proběhnout dostatečně dlouhým cyklem, aby nastaly podmínky pro selhání.

Tento test určitě vytvoří podmínky, aby vznikly problémy se souběžnou aktualizací. To je však tak řídké, že ve většině výskytů to tento test nedokáže detekovat.

Skutečně, abychom tento problém doopravdy dokázali detektovat, potřebujeme nastavit počet iterací na více než milion. I tak během deseti spuštění a s cyklem jeden milion nastal tento problém jen jednou. To znamená, že bychom zřejmě měli nastavit iteraci nad sto milionů, aby program skutečně selhal. Jako dlohu jsme ochotni čekat?

I kdybychom vyladili test pro určitá selhání na jednom počítači, museli bychom tento test pravděpodobně přeladit na jiné hodnoty a demonstrovat selhání na jiném počítači, jiném operačním systému nebo jiné verzi JVM.

A tohle je jednoduchý problém. Pokud nejsme schopni demonstrovat chybný kód s tímto problémem nějak jednoduše, jak budeme vůbec detektovat skutečně komplikované problémy?

Takže, jaký použijeme postup, abychom demonstrovali jednoduché selhání? A co je důležitější, jak máme napsat testy, které by nám ukázaly selhání v komplikovanějším kódu? Jak můžeme zjistit, zda nás kód nemá chyby, když ani nevíme, kde hledat?

Zde je několik nápadů:

- ◆ **Testování metodou Monte Carlo.** Umožňuje pružné testy, které lze vyladit. Pak tento test spouštějte neustále dokola – řekněme na testovacím serveru – a náhodně měňte ladící hodnoty. Pokud testy někdy neprojdou, kód je chybný. Začněte s psaním testů brzy, aby je integrační server mohl co nejdříve začít souvisle spouštět. Mimořádne, dbejte na to, abyste pečlivě zaprotokolovali podmínky, za kterých test selhal.
- ◆ Testy spouštějte na každé cílové platformě, na kterou budete kód instalovat. Opakováně. Nepřetržitě. Čím déle poběží testy bez selhání, tím je pravděpodobnější, že:
  - ostrý kód neobsahuje chyby nebo
  - testy nejsou dostatečné, aby nějaký problém našly.
- ◆ Spouštějte testy na počítači s proměnlivou zátěží. Pokud umíte simulovat zatížení podobné ostatnímu prostředí, využijte toho.

Ale i když uděláte vše, co jsme si ukázali, stále ještě nemáte příliš velikou šanci nalézt v kódu problémy s podprocesy. Nejzákeřnější problémy jsou ty, které mají tak účinný průřez, že mohou nastat jedenkrát za miliardu pokusů. Takové problémy jsou noční můrou komplikovaných systémů.

## Podpora nástrojů pro testování kódu s podprocesy

Firma IBM vytvořila nástroj jménem ConTest<sup>6</sup>. Upravuje třídy tak, aby bylo pravděpodobnější, že nezabezpečený kód s podprocesy selže.

Nemáme žádné přímé vztahy s firmou IBM nebo s týmem, který vyvinul nástroj ConTest. Jeden z našich kolegů nás na něj upozornil. Po několika minutách, co jsme tento nástroj používali, jsme zjistili, že umíme nacházet problémy v podprocesech mnohem lépe.

Zde je pár poznámek, jak používat ConTest:

6. <http://www.haifa.ibm.com/projects/verification/contest/index.html>.

- ◆ Napište testy a ostrý kód a přesvědčte se, že jsou mezi nimi testy navržené speciálně pro simulaci více uživatelů a s různou zátěží, jak jsme se již zmínili výše.
- ◆ Vybavte testy a ostrý kód nástrojem ConTest.
- ◆ Spusťte testy.

Když jsme kód vybavili nástrojem ConTest, míra našeho úspěchu stoupla přibližně z jednoho selhání za deset milionů iterací na jedno selhání na *třicet* iterací. Zde jsou hodnoty cyklů pro několik běhů testu po jeho úpravě: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Takže je jasné, že upravené třídy selhávaly mnohem dříve a s mnohem větší spolehlivostí.

## Závěr

Tato kapitola pobývala velmi krátce ve velkém a zrádném území souběžného programování. Sotva jsme se dotkli povrchu. Nás důraz byl kladen na postupy, jak udržet souběžný kód čistý, ale je toho ještě velmi mnoho, co byste se měli učit, pokud se chystáte psát souběžné systémy. Doporučujeme vám začít s vynikající knihou *Concurrent Programming in Java: Design Principles and Patterns*<sup>7</sup>, jejím autorem je Doug Lea.

V této kapitole jsme hovořili o souběžné aktualizaci a postupech čisté synchronizace a zamykání, které umožňuje, abychom se těmto problémům vyhnuli. Hovořili jsme o tom, jak mohou podprocesy zvýšit propustnost systému, založeného na vstupních a výstupních operacích, a ukázali jsme si čisté techniky, kterými můžeme takovýcho pokroků dosáhnout. Hovořili jsme o zablokování a postupech, jak tomu zabránit čistým způsobem. Nakonec jsme se dostali ke strategiím pro odhalování problémů souběžného kódu tím, že jej vybavíme vhodnými nástroji.

## Výukový program

### Klient/server bez podprocesů

#### Výpis A.3. 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("Server Starting\n");  
    }
```

7. Viz [Lea99] str. 202.

```
while (keepProcessing) {
    try {
        System.out.printf("accepting client\n");
        Socket socket = serverSocket.accept();
        System.out.printf("got client\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("Server: getting message\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Server: got message: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Server: sending reply: %s\n", message);
        MessageUtils.sendMessage(socket, "Processed: " + message);
        System.out.printf("Server: sent\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) {
        }
}
```

**Výpis A.4. 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("Server Starting\n");
        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\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("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\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) {
        }
}
}
}

```

#### Výpis A.5. 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/server s podprocesy

Změnit server tak, aby používal podprocesy, vyžaduje jen změnit zprávu od procesu (nové řádky jsou zvýrazněné, aby byly lépe vidět):

```

void process(final Socket socket) {
    if (socket == null)
        return;
    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                System.out.printf("Server: getting message\n");

```

```
String message = MessageUtils.getMessage(socket);
System.out.printf("Server: got message: %s\n", message);
Thread.sleep(1000);
System.out.printf("Server: sending reply: %s\n", message);
MessageUtils.sendMessage(socket, "Processed: " + message);
System.out.printf("Server: sent\n");
closeIgnoringException(socket);
} catch (Exception e) {
    e.printStackTrace();
}
};

Thread clientConnection = new Thread(clientHandler);
clientConnection.start();
}
```



# DODATEK B

org.jfree.date.SerialDate

**Výpis B.1. SerialDate.java**

```
1 /* =====
2 * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
3 * =====
4 *
5 * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
6 *
7 * Informace o projektu: http://www.jfree.org/jcommon/index.html
8 *
9 * Tato knihovna je volný software; můžete ji šířit a modifikovat
10 * za podmínek GNU Lesser General Public License, jak byla publikována
11 * organizací Free Software Foundation; bud' verze 2.1 licence, nebo
12 * (podle vaší volby) jakákoli pozdější verze.
13 *
14 * Tato knihovna je rozšiřována a očekává se, že bude užitečná, ale
15 * BEZ JAKÉKOLIV ZÁRUKY; dokonce i bez implikované záruky z důvodu UVEDENÍ
16 * NA TRH nebo VHODNOSTI PRO URČITÝ ÚČEL. Viz GNU Lesser General Public
17 * License, kde naleznete další detaily.
18 *
19 * Kopii licence GNU Lesser General Public License byste měli obdržet
20 * spolu s touto knihovnou; pokud ne, obratte se na organizaci Free
21 * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
22 * 02110- 1301,
23 * USA.
24 *
25 * [Java je obchodní značka nebo registrovaná obchodní značka Sun Microsystems,
26 * Inc.
27 * ve Spojených státech a jiných zemích.]
28 *
29 * -----
30 * SerialDate.java
31 * -----
32 * (C) Copyright 2001-2005, Object Refinery Limited.
33 *
34 * Původní autor: David Gilbert (Object Refinery Limited);
35 * Přispěvatelé: -;
36 *
37 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
38 *
39 * Změny (od 11. října 2001)
40 * -----
41 * 11. říjen 2001 : Reorganizace třídy a její přesun do nového balíčku
42 * com.jrefinery.date (DG);
43 * 05. listopad 2001 : Přidána metoda getDescription(), a odstraněna třída
44 * NotableDate (DG);
45 * 12. listopad 2001 : IBD vyžaduje metodu setDescription(), když je nyní
46 * odstraněna třída NotableDate (DG); změna v getPreviousDayOfWeek(),
47 * getFollowingDayOfWeek() a getNearestDayOfWeek() a opraveny
48 * chyby (DG);
49 * 05. prosinec 2001 : Opravena chyba ve třídě SpreadsheetDate (DG);
50 * 29. květen 2002 : Přesunuty konstanty měsíců do samostatného rozhraní
51 * (MonthConstants) (DG);
```

```
50 * 27. srpen 2002 : Opraveny chyby v metodách addMonths(), díky Petru Nálevkovi
51 * (DG);
52 * 03. říjen 2002 : Opraveny chyby, které oznámil Checkstyle (DG);
53 * 13. březen 2003 : Implementace Serializable (DG);
54 * 29. květen 2003 : Opravena chyba v metodě addMonths (DG);
55 * 04. září 2003 : Implementace Comparable. Aktualizace isInRange javadocs
56 * (DG);
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 * Abstraktní třída, která definuje naše požadavky pro zpracování dat,
69 * aniž by svazovala konkrétní implementaci.
70 * <P>
71 * Požadavek 1 : je nutné alespoň vyhovět požadavkům zpracování dat v Excelu
72 * Požadavek 2 : třída je neměnná;
73 * <P>
74 * Proč nepoužívat java.util.Date? Ano, až to bude mít smysl. java.util.Date
75 * může být zatím *příliš* precizní - reprezentuje instanci v čase,
76 * s přesností jedné tisíciny sekundy (samotné datum závisí na
77 * časovém pásmu). Někdy jen potřebujeme znázornit konkrétní den (např. 21
78 * ledna 2015), aniž by nás zajímal čas, nebo
79 * časové pásmo nebo něco jiného. To je důvod, proč jsme definovali SerialDate.
80 * <P>
81 * Můžete volat getInstance() a obdržet konkrétní podtřídu SerialDate,
82 * a nemusíte se starat o přesnou implementaci.
83 *
84 * @author David Gilbert
85 */
86 public abstract class SerialDate implements Comparable,
87                                         Serializable,
88                                         MonthConstants {
89
90     /** Pro serializaci. */
91     private static final long serialVersionUID = -293716040467423637L;
92
93     /** Symboly datových formátů. */
94     public static final DateFormatSymbols
95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
96
97     /** Sériové číslo pro 1. ledna 1900. */
98     public static final int SERIAL_LOWER_BOUND = 2;
99 }
```

```
100     /** Sériové číslo pro 31. prosinec 9999. */
101    public static final int SERIAL_UPPER_BOUND = 2958465;
102
103    /** Nejnižší hodnota roku podporovaná datovým formátem. */
104    public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106    /** Nejvyšší hodnota roku podporovaná datovým formátem. */
107    public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109    /** Konstanta pro pondělí. Ekvivalentní s java.util.Calendar.MONDAY. */
110    public static final int MONDAY = Calendar.MONDAY;
111
112    /**
113     * Užitečná konstanta pro úterý. Ekvivalentní s java.util.Calendar.TUESDAY.
114     */
115    public static final int TUESDAY = Calendar.TUESDAY;
116
117    /**
118     * Konstanta pro středu. Ekvivalentní s java.util.Calendar.WEDNESDAY.
119     */
120    public static final int WEDNESDAY = Calendar.WEDNESDAY;
121
122
123    /**
124     * Konstanta pro čtvrtok. Ekvivalentní s java.util.Calendar.THURSDAY.
125     */
126    public static final int THURSDAY = Calendar.THURSDAY;
127
128    /** Konstanta pro pátek. Ekvivalentní s java.util.Calendar.FRIDAY. */
129    public static final int FRIDAY = Calendar.FRIDAY;
130
131    /**
132     * Konstanta pro sobotu. Ekvivalentní s java.util.Calendar.SATURDAY.
133     */
134    public static final int SATURDAY = Calendar.SATURDAY;
135
136    /** Konstanta pro neděli. Ekvivalentní s java.util.Calendar.SUNDAY. */
137    public static final int SUNDAY = Calendar.SUNDAY;
138
139    /** Počet dnů v každé měsíc v nepřestupných letech. */
140    static final int[] LAST_DAY_OF_MONTH =
141        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143    /** Počet dnů v nepřestupném roce až do konce měsíce. */
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    /** Počet dnů v roce do konce předchozího měsíce. */
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    /** Počet dnů v přestupném roce do konce měsíce. */
```

```
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      * Počet dnů v přestupném roce do konce předchozího měsíce.
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     /**
163      * Užitečná konstanta pro odkaz na první týden v měsíci.
164      */
165     public static final int FIRST_WEEK_IN_MONTH = 1;
166
167     /**
168      * Užitečná konstanta pro odkaz na druhý týden v měsíci.
169      */
170     public static final int SECOND_WEEK_IN_MONTH = 2;
171
172     /**
173      * Užitečná konstanta pro odkaz na třetí týden v měsíci.
174      */
175     public static final int THIRD_WEEK_IN_MONTH = 3;
176
177     /**
178      * Užitečná konstanta pro odkaz na čtvrtý týden v měsíci.
179      */
180     public static final int FOURTH_WEEK_IN_MONTH = 4;
181
182     /**
183      * Užitečná konstanta pro odkaz na poslední týden v měsíci.
184      */
185     public static final int LAST_WEEK_IN_MONTH = 0;
186
187     /**
188      * Užitečná konstanta pro rozsah.
189      */
190     public static final int INCLUDE_NONE = 0;
191
192     /**
193      * Užitečná konstanta pro rozsah.
194      */
195     public static final int INCLUDE_FIRST = 1;
196
197     /**
198      * Užitečná konstanta pro rozsah.
199      */
200     public static final int INCLUDE_SECOND = 2;
201
202     /**
203      * Užitečná konstanta pro rozsah.
204      */
205     public static final int INCLUDE_BOTH = 3;
206
207     /**
208      * Užitečná konstanta pro specifikaci dne v týdnu, relativně vůči pevnému
209      * datu.
210      */
211     public static final int PRECEDING = -1;
212
213     /**
214      * Užitečná konstanta pro specifikaci dne v týdnu relativně vůči pevnému
215      * datu.
216      */
217     public static final int NEAREST = 0;
218
219     /**
220      * Užitečná konstanta pro specifikaci dne v týdnu relativně vůči pevnému
221      * datu.
222      */
223
```

```
204     */
205     public static final int FOLLOWING = 1;
206
207     /** Popis data. */
208     private String description;
209
210     /**
211      * Implicitní konstruktor.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Vrací <code>true</code>, pokud je předaný celočíselný kód, reprezentuje
218      * platný den v týdnu. Jinak vrací <code>false</code>.
219      *
220      * @param code kontrolovaný kód, zda je platný.
221      *
222      * @return vrací <code>true</code>, pokud je předávaný kód, reprezentuje
223      * platný den v týdnu. Jinak vrací <code>false</code>.
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      * Převede předaný řetězec na den v týdnu.
243      *
244      * @param s řetězec, reprezentující den v týdnu.
245      *
246      * @return vrací <code>-1</code>, pokud nelze řetězec převést,
247      * jinak vrací týden.
248      */
249     public static int stringToWeekdayCode(String s) {
250
251         final String[] shortWeekdayNames
252             = DATE_FORMAT_SYMBOLS.getShortWeekdays();
253         final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
254
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 * Vrací řetězec reprezentující předaný den v týdnu.
273 * <P>
274 * Je třeba hledat lepší přístup.
275 *
276 * @param weekday den v týdnu.
277 *
278 * @return vrací řetězec, reprezentující předaný den v týdnu.
279 */
280 public static String weekdayCodeToString(final int weekday) {
281
282     final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
283     return weekdays[weekday];
284
285 }
286 /**
287 * Vrací pole jmen měsíců.
288 *
289 * @return vrací pole jmen měsíců.
290 */
291 public static String[] getMonths() {
292
293     return getMonths(false);
294
295 }
296
297 /**
298 * Vrací pole jmen měsíců.
299 *
300 * @param zkrácený příznak indikující, že se mají vracet zkrácená
301 * jména měsíců.
302 *
303 * @return Vrací pole jmen měsíců.
304 */
305 public static String[] getMonths(final boolean shortened) {
```

```
308     if (shortened) {
309         return DATE_FORMAT_SYMBOLS.getShortMonths();
310     }
311     else {
312         return DATE_FORMAT_SYMBOLS.getMonths();
313     }
314 }
315
316 /**
317 * Vrací hodnotu true, reprezentuje-li předaný celočíselný kód platný
318 * měsíc.
319 *
320 * @param code kód, kontrolovaný na platnost.
321 *
322 * @return vrací <code>true</code>, pokud předaný celočíselný kód
323 * reprezentuje platný měsíc.
324 */
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 * Vrací čtvrtletí daného měsíce.
350 *
351 * @param code kód měsíce (1-12).
352 *
353 * @return vrací čtvrtletí, do kterého spadá měsíc.
354 * @throws vyvolává výjimku 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 * Vrací řetězec, který reprezentuje předávaný měsíc.
378 * <P>
379 * Vrácený řetězec je dlouhou variantou názvu měsíce, převzatého
380 * z implicitního národního prostředí.
381 *
382 *
383 * @param month měsíc.
384 *
385 * @return vrací řetězec reprezentující předávaný měsíc.
386 */
387 public static String monthCodeToString(final int month) {
388
389     return monthCodeToString(month, false);
390 }
391 }
392 /**
393 * Vrací řetězec reprezentující předávaný měsíc.
394 * <P>
395 * Návratová hodnota řetězce je krátkou nebo dlouhou verzi měsíce podle
396 * implicitního národního prostředí.
397 *
398 *
399 * @param month měsíc.
400 * @param zkrácená verze měsíce, vrací-li se hodnota <code>true</code>.
401 *
402 *
403 * @return řetězec, reprezentující předávaný měsíc.
404 * @throws vyvolá výjimku 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 * Převede řetězec na kód měsíce.
430 * <P>
431 * Tato metoda vrátí jednu z konstant JANUARY, FEBRUARY, ....
432 * DECEMBER, které odpovídají danému řetězci. Pokud nelze řetězec
433 * rozpoznat, vrací metoda -1.
434 *
435 * @param s analyzovaný řetězec.
436 *
437 * @return vrací hodnotu <code>-1</code>, pokud řetězec nelze
438 * analyzovat, jinak vrací měsíc v roce.
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     // nejdříve zkus analyzovat řetězec jako celé číslo (1-12)...
449     try {
450         result = Integer.parseInt(s);
451     }
452     catch (NumberFormatException e) {
453         // suppress
454     }
455
456     // nyní hledejte mezi názvy měsíce...
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         }
464     }
465 }
```

```
463             if (s.equals(monthNames[i])) {
464                 result = i + 1;
465                 break;
466             }
467         }
468     }
469
470     return result;
471 }
472 }
473
474 /**
475 * Vrací hodnotu true, pokud dodaný celočíselný kód reprezentuje platný
476 * týden v měsíci, jinak vrací false.
477 *
478 * @param code kód, kontrolovaný na platnost.
479 * @return vrací hodnotu <code>true</code>, pokud dodaný celočíselný kód
480 * reprezentuje platný týden v měsíci.
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 * Určuje, zda daný rok je či není přestupný.
497 *
498 * @param yyyy rok (v rozsahu 1900 až 9999).
499 *
500 * @return vrací <code>true</code>, pokud je daný rok přestupný.
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 }
```

```
515         }
516
517     }
518     *
519     /**
520      * Vrací počet přestupných let v intervalu od 1900 až po daný rok
521      * VČETNĚ.
522      * <P>
523      * Poznamenejme, že rok 1900 není přestupný.
524      *
525      * @param yyyy rok (v rozsahu od 1900 do 9999).
526      *
527      * @return počet přestupných let od roku 1900 až po daný rok.
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      * Vrací číslo posledního dne v měsíci, v úvahu se berou
540      * přestupné roky.
541      *
542      * @param month měsíc.
543      * @param yyyy rok (v rozsahu od 1900 do 9999).
544      *
545      * @return číslo posledního dne v měsíci.
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      * Vytvoří nové datum tím, že k základnímu datu přičte určitý počet dnů.
564      *
565      *
566      * @param days počet dnů, které se mají přičíst (může být i záporný).

```

```
567     * @param base základní datum.  
568     *  
569     * @return nové datum.  
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      * Vytváří nové datum. K základnímu datu se přičte určitý  
579      * počet měsíců.  
580      * <P>  
581      * Pokud je základní datum blízko konce měsíce, výsledný den  
582      * může být nepatrně upraven: 31 květen + 1 měsíc = 30 červen.  
583      *  
584      * @param months počet měsíců, které se mají přičíst (může být i záporný).  
585      * @param base základní datum.  
586      *  
587      * @return nové datum.  
588      */  
589     public static SerialDate addMonths(final int months,  
590         final SerialDate base) {  
591  
592         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)  
593             / 12;  
594         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)  
595             % 12 + 1;  
596         final int dd = Math.min(  
597             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)  
598         );  
599         return SerialDate.createInstance(dd, mm, yy);  
600     }  
601  
602     /**  
603      * Vytvoří nové datum přičtením určitého počtu let k základnímu  
604      * datu.  
605      *  
606      * @param years počet let, které se mají přičíst (může být i negativní).  
607      * @param base základní datum.  
608      *  
609      * @return nové datum.  
610      */  
611     public static SerialDate addYears(final int years, final SerialDate base) {  
612  
613         final int baseY = base.getYYYY();  
614         final int baseM = base.getMonth();  
615         final int baseD = base.getDayOfMonth();  
616  
617         final int newYear = baseY + years;  
618         final int newMonth = (baseM + years * 12) % 12 + 1;
```

```
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 * Vrací poslední datum, které vyhovuje danému dni v týdnu
630 * a nastává PŘED základním datem.
631 *
632 * @param targetWeekday kód pro cílový den v týdnu.
633 * @param base základní datum.
634 *
635 * @return poslední datum, které vyhovuje danému dni v týdnu
636 * a nastává PŘED základním datem.
637 */
638 public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639 final SerialDate base) {
640
641     // zkонтролуй аргументы...
642     if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643         throw new IllegalArgumentException(
644             "Invalid day-of-the-week code."
645         );
646     }
647
648     // najdi datum...
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 * Vrací první datum, které vyhovuje danému dni v týdnu
664 * a nastává PO základním datu.
665 *
666 * @param targetWeekday kód pro cílový den v týdnu.
667 * @param base základní datum.
668 *
669 * @return první datum, které vyhovuje danému dni v týdnu
670 * a nastává po základním datu.
```

```
671     */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                         final SerialDate base) {
674
675         // zkontroluj argumenty...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code."
679             );
680         }
681
682         // vyhledej datum...
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      * Vrací datum, které vyhovuje danému dni v týdnu a je
697      * NEJBLÍŽE základnímu datu.
698      *
699      * @param targetDOW kód pro cílový den v týdnu.
700      * @param base základní datum.
701      *
702      * @return vrací datum, které vyhovuje danému dni v týdnu a je
703      * NEJBLÍŽE základnímu datu.
704      */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                               final SerialDate base) {
707
708         // zkontroluj argumenty...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code."
712             );
713         }
714
715         // najdi datum...
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      * Posune datum vpřed na poslední den v měsíci.
730      *
731      * @param base základní datum.
732      *
733      * @return vrací sériové datum.
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      * Vrací řetězec odpovídající kódu týdne v měsíci.
744      * <P>
745      * Je potřeba nalézt lepší přístup.
746      *
747      * @param count celočíselný kód reprezentující týden v měsíci.
748      *
749      * @return vrací řetězec odpovídající kódu pro týden v měsíci.
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      * Vrací řetězec reprezentující předané "příbuzné".
767      * <P>
768      * Je potřeba nalézt lepší přístup.
769      *
770      * @param relative konstanta, reprezentující "příbuzného".
771      *
772      * @return řetězec, reprezentující dodaného "příbuzného".
773     */
774     public static String relativeToString(final int relative) {
```

```
775     switch (relative) {
776         case SerialDate.PRECEDING : return "Preceding";
777         case SerialDate.NEAREST : return "Nearest";
778         case SerialDate.FOLLOWING : return "Following";
779         default : return "ERROR : Relative To String";
780     }
781 }
782 }
783 /**
784 * Tovární metoda, která vrací instanci konkrétní podtřídy třídy
785 * {@link SerialDate}.
786 *
787 * @param day den (1-31).
788 * @param month měsíc (1-12).
789 * @param yyyy rok (v rozsahu od 1900 do 9999).
790 *
791 * @return instance třídy {@link SerialDate}.
792 */
793 public static SerialDate createInstance(final int day, final int month,
794                                         final int yyyy) {
795     return new SpreadsheetDate(day, month, yyyy);
796 }
797 /**
798 * Tovární metoda, která vrací instanci konkrétní podtřídy třídy
799 * {@link SerialDate}.
800 *
801 * @param serial sériové číslo dne (1 Leden 1900 = 2).
802 *
803 * @return instance třídy SerialDate.
804 */
805 public static SerialDate createInstance(final int serial) {
806     return new SpreadsheetDate(serial);
807 }
808 /**
809 * Tovární metoda, která vrací instanci podtřídy SerialDate.
810 *
811 * @param date datový objekt jazyka Java.
812 *
813 * @return instance třídy SerialDate.
814 */
815 public static SerialDate createInstance(final java.util.Date date) {
816
817     final GregorianCalendar calendar = new GregorianCalendar();
818     calendar.setTime(date);
819     return new SpreadsheetDate(calendar.get(Calendar.DATE),
820                               calendar.get(Calendar.MONTH) + 1,
821                               calendar.get(Calendar.YEAR));
822
823 }
```

```
827     }
828
829     /**
830      * Vrací sériové číslo data, které je pro 1.leden 1900 = 2 (což
831      * téměř odpovídá numerickému systému, který používá Microsoft Excel pro
832      * Windows a Lotus 1-2-3).
833      *
834      * @return vrací sériové číslo data.
835      */
836     public abstract int toSerial();
837
838     /**
839      * Vrací java.util.Date. Protože má java.util.Date vyšší přesnost než
840      * SerialDate, potřebujeme definovat pravidlo pro "čas během dne".
841      *
842      * @return this jako <code>java.util.Date</code>.
843      */
844     public abstract java.util.Date toDate();
845
846     /**
847      * Vrací popis data.
848      *
849      * @return popis data.
850      */
851     public String getDescription() {
852         return this.description;
853     }
854
855     /**
856      * Nastavuje popis data.
857      *
858      * @param description nový popis data.
859      */
860     public void setDescription(final String description) {
861         this.description = description;
862     }
863
864     /**
865      * Převádí datum na řetězec.
866      *
867      * @return vrací znakovou reprezentaci data.
868      */
869     public String toString() {
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871                         + "-" + getYYYY();
872     }
873
874     /**
875      * Vrací rok (předpokládáme platný rozsah od 1900 do 9999).
876      *
877      * @return vrací rok.
878      */
```

```
879     public abstract int getYYYY();  
880  
881     /**  
882      * Vrací měsíc (leden = 1, únor = 2, březen = 3).  
883      *  
884      * @return vrací měsíc v roce.  
885      */  
886     public abstract int getMonth();  
887  
888     /**  
889      * Vrací den v měsíci.  
890      *  
891      * @return vrací den v měsíci.  
892      */  
893     public abstract int getDayOfMonth();  
894  
895     /**  
896      * Vrací den v měsíci.  
897      *  
898      * @return vrací den v měsíci.  
899      */  
900     public abstract int getDayOfWeek();  
901  
902     /**  
903      * Vrací rozdíl (ve dnech) mezi tímto datem a určitým  
904      * "jiným" datem.  
905      * <P>  
906      * Výsledek je kladný, pokud tohle datum následuje po "jiném" datu,  
907      * a záporný, pokud "jinému" datu předchází.  
908      *  
909      * @param jiné datum, se kterým srovnáváme.  
910      *  
911      * @return vrací rozdíl mezi tímto datem a jiným datem.  
912      */  
913     public abstract int compare(SerialDate other);  
914  
915     /**  
916      * Vrací hodnotu true, jestliže aktuální instance třídy SerialDate ,  
917      * reprezentuje stejné datum jako předaný parametr.  
918      *  
919      * @param jiné datum, se kterým se porovnává.  
920      *  
921      * @return vrací <code>true</code>, jestliže aktuální instance třídy  
922      * SerialDate reprezentuje jiné datum, než specifikuje parametr.  
923      */  
924     public abstract boolean isOn(SerialDate other);  
925  
926     /**  
927      * Vrací hodnotu true, pokud aktuální instance třídy SerialDate reprezentuje  
928      * dřívější datum než parametr.  
929      *  
930      * @param other Datum, se kterým porovnáváme.
```

```
931      *
932      * @return vrací <code>true</code>, pokud aktuální SerialDate reprezentuje
933      * dřívější datum, než parametr.
934      */
935      public abstract boolean isBefore(SerialDate other);
936
937      /**
938      * Vrací hodnotu true, pokud aktuální instance reprezentuje stejné datum
939      * jako parametr.
940      *
941      * @param other Datum, se kterým porovnáváme.
942      *
943      * @return vrací <code>true</code>, pokud aktuální instance
944      * reprezentuje stejné datum jako parametr.
945      */
946      public abstract boolean isOnOrBefore(SerialDate other);
947
948      /**
949      * Vrací hodnotu true, pokud tato třída SerialDate reprezentuje stejné
950      * datum,
951      * jako specifikovaná SerialDate.
952      *
953      * @param other Datum, se kterým probíhá porovnání.
954      *
955      * @return vrací <code>true</code>, pokud tato třída SerialDate
956      * reprezentuje stejné datum, jako specifikovaná SerialDate.
957      */
958      public abstract boolean isAfter(SerialDate other);
959
960      /**
961      * Vrací hodnotu true, pokud aktuální instance reprezentuje stejné datum
962      * jako parametr.
963      *
964      * @param other Datum, se kterým porovnáváme.
965      *
966      * @return vrací <code>true</code>, pokud aktuální instance
967      * reprezentuje stejné datum jako parametr.
968      */
969      public abstract boolean isOnOrAfter(SerialDate other);
970
971      /**
972      * Vrací hodnotu <code>true</code>, pokud {@link SerialDate} spadá do
973      * daného intervalu (VČETNĚ). Pořadí dat d1 a d2 není
974      * důležité.
975      *
976      * @param d1 hranice datového intervalu.
977      * @param d2 druhá hranice datového intervalu.
978      *
979      * @return A boolean.
980      */
981      public abstract boolean isInRange(SerialDate d1, SerialDate d2);
```

```
982     /**
983      * Vrací hodnotu <code>true</code>, pokud aktuální instance {@link
984      * SerialDate} spadá do daného intervalu (volající objekt určuje, zda májí
985      * být hranice do intervalu zahrnuty či ne). Na pořadí dat d1 a d2 nezáleží.
986      *
987      * @param d1 jedna hranice datového intervalu.
988      * @param d2 druhá hranice datového intervalu.
989      * @param include Kód, který určuje, zda budou hranice
990      * do intervalu zahrnuty či ne.
991      *
992      * @return A boolean.
993      */
994     public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995         int include);
996
997     /**
998      * Vrací poslední datum, které odpovídá určenému dni v týdnu a nastává
999      * PŘED tímto datem.
1000     *
1001     * @param targetDOW Kód cílového dne v týdnu.
1002     *
1003     * @return vrací poslední datum, která odpovídá určenému dni v týdnu
1004     * a nastává PŘED tímto datem.
1005     */
1006     public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007         return getPreviousDayOfWeek(targetDOW, this);
1008     }
1009
1010    /**
1011     * Vrací nejstarší datum, které odpovídá určenému dni v týdnu
1012     * a nastává PO tomto datu.
1013     *
1014     * @param targetDOW Kód, který určuje cílový den v týdnu.
1015     *
1016     * @return vrací první datum, které odpovídá určenému dni v týdnu
1017     * a nastává PO tomto datu.
1018     */
1019     public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020         return getFollowingDayOfWeek(targetDOW, this);
1021     }
1022
1023    /**
1024     * Vrací nejbližší datum, které odpovídá určenému dni v týdnu.
1025     *
1026     * @param targetDOW Kód pro určení cílového dne v týdnu.
1027     *
1028     * @return vrací nejbližší datum, které odpovídá určenému dni v týdnu.
1029     */
1030     public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031         return getNearestDayOfWeek(targetDOW, this);
1032     }
1033
1034 }
```

**Výpis B.2. SerialDateTest.java**

```
1  /*
2  * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
3  *
4  *
5  * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
6  *
7  * Informace o projektu: http://www.jfree.org/jcommon/index.html
8  *
9  * Tato knihovna je volný software; můžete ji šířit a modifikovat
10 * za podmínek GNU Lesser General Public License, jak byla publikovaná
11 * organizací Free Software Foundation; buď verze 2.1 licence, nebo
12 * (podle vaši volby) jakákoli pozdější verze.
13 *
14 * Tato knihovna je rozšiřována a očekává se, že bude užitečná, ale
15 * BEZ JAKÉKOLIV ZÁRUKY; dokonce i bez implikované záruky z důvodu UVEDENÍ
16 * NA TRH nebo VHODNOSTI PRO URČITÝ ÚČEL. Viz GNU Lesser General Public
17 * License, kde naleznete další detaily.
18 *
19 * Kopii licence GNU Lesser General Public License byste měli obdržet
20 * spolu s touto knihovnou; pokud ne, obrátte se na organizaci Free
21 * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22 * MA 02110-1301,USA.
23 *
24 * [Java je obchodní značka nebo registrovaná obchodní značka
25 * Sun Microsystems, Inc. ve Spojených státech a jiných zemích.]
26 *
27 * -----
28 * SerialDate.java
29 * -----
30 * (C) Copyright 2001-2005, Object Refinery Limited.
31 *
32 * Původní autor: David Gilbert (Object Refinery Limited);
33 * Přispěvatelé: -;
34 *
35 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
36 *
37 * Změny
38 * -----
39 * 15-Listopad-2001 : Verze 1 (DG);
40 * 25-Červen-2002 : Odstraněn zbytečný import (DG);
41 * 24-Říjen-2002 : Opraveny chyby, které oznámil Checkstyle (DG);
42 * 13-Březen-2003 : Přidán test serializace (DG);
43 * 05-Leden-2005 : Přidán test pro sestavy chyb 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  * Některé testy JUnit pro třídu {@link SerialDate}.
65  */
66 public class SerialDateTests extends TestCase {
67
68     /** Datum, reprezentující 9. listopad */
69     private SerialDate nov9Y2001;
70
71     /**
72      * Vytvoří nový testovací případ.
73      *
74      * @param name jméno.
75      */
76     public SerialDateTests(final String name) {
77         super(name);
78     }
79
80     /**
81      * Vrací testovací sadu pro běhové prostředí JUnit.
82      *
83      * @return vrací testovací sadu.
84      */
85     public static Test suite() {
86         return new TestSuite(SerialDateTests.class);
87     }
88
89     /**
90      * Problém je nastaven.
91      */
92     protected void setUp() {
93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER,
94             2001);
95     }
96
97     /**
98      * 9. listopad 2001 plus dva měsíce by mělo dát 9. leden 2002.
99      */
100    public void testAddMonthsTo9Nov2001() {
101        final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
102        final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
103        assertEquals(answer, jan9Y2002);
```

```
103    }
104
105    /**
106     * Testovací sada pro nahlášenou chybu, nyní opraveno.
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     * Testovací sada pro nahlášenou chybu, nyní opraveno.
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     * Pondělí předcházející pátku 9. listopadu 2001 by mělo být 5. listopadu.
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     * Pondělí následující po pátku 9. listopadu 2001 by měl být 12. listopad.
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     * Pondělí nejbližše pátku 9. listopadu 2001 by mělo být 12. listopad.
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     * Nejbližší pondělí k 22. lednu 1970 připadá na 19.
```

```
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      * Problém: převod dnů na řetězce vrací správné hodnoty. Ve skutečnosti
164      * závisejí na národním nastavení, takže tento test je třeba modifikovat.
165      */
166     public void testWeekdayCodeToString() {
167
168         final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169         assertEquals("Saturday", test);
170     }
171
172     /**
173      * Test převodu řetězce na den v týdnu. Selže, pokud implicitní národní
174      * nastavení nepoužívá anglické názvy dnů...je třeba vymyslet lepší test
175      */
176     public void testStringToWeekday() {
177
178         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
179         assertEquals(SerialDate.WEDNESDAY, weekday);
180
181         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
182         assertEquals(SerialDate.WEDNESDAY, weekday);
183
184         weekday = SerialDate.stringToWeekdayCode("Wed");
185         assertEquals(SerialDate.WEDNESDAY, weekday);
186
187     }
188
189     /**
190      * Test převodu řetězce na měsíc. Test selže, pokud implicitní národní
191      * nastavení nepoužívá anglické názvy pro měsice...třeba vymyslet lepší
192      * 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      * Testuje převod kódu měsíce na řetězec.
209      */
210     public void testMonthCodeToStringCode() {
211
212         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213         assertEquals("December", test);
214
215     }
216
217     /**
218      * Rok 1900 není přestupný.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * Rok 2000 je přestupný.
226      */
227     public void testIsLeapYear2000() {
228         assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * Počet přestupných roků od roku 1900 do 1899 včetně je roven 0.
233      */
234     public void testLeapYearCount1899() {
235         assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239      * Počet přestupných roků od 1900 do 1903 včetně je roven 0.
240      */
241     public void testLeapYearCount1903() {
242         assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * Počet přestupných roků od roku 1900 do 1904 včetně je roven 1.
247      */
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253      * Počet přestupných roků od roku 1900 do 1999 včetně je 24.
254      */
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
```

```
258
259     /**
260      * Počet přestupných roků od roku 1900 do 2000 včetně je 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Serializuj instanci, obnov ji a zkontroluj, zda se shodují.
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             ObjectOutputStream out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInputStream in = new ObjectInputStream(
281                             ByteArrayInputStream(buffer.toByteArray()));
282             d2 = (SerialDate) in.readObject();
283             in.close();
284         } catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288     }
289
290     /**
291      * Test pro sestavu chyb 1096282 (opraveno).
292      */
293     public void test1096282() {
294         SerialDate d = SerialDate.createInstance(29, 2, 2004);
295         d = SerialDate.addYears(1, d);
296         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
297         assertTrue(d.isOn(expected));
298     }
299
300     .
301     /**
302      * Různé testy metody addMonths().
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 }
```

#### Výpis B.3. MonthConstants.java

```
1 /**
2 * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
3 * -----
4 *
5 * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
6 *
7 * Informace o projektu: http://www.jfree.org/jcommon/index.html
8 *
9 * Tato knihovna je volný software; můžete ji šířit a modifikovat
10 * za podmínek GNU Lesser General Public License, jak byla publikována
11 * organizací Free Software Foundation; buď verze 2.1 licence, nebo
12 * (podle vaší volby) jakákoli pozdější verze.
13 *
14 * Tato knihovna je šířena a očekává se, že bude užitečná, ale
15 * BEZ JAKÉKOLIV ZÁRUKY; dokonce i bez implikované záruky z důvodu UVEDENÍ
16 * NA TRH nebo VHODNOSTI PRO URČITÝ ÚČEL. Viz GNU Lesser General Public
17 * Licence kde najeznete další detaily.
18 *
19 * Kopii licence GNU Lesser General Public License byste měli obdržet
20 * spolu s touto knihovnou; pokud ne, obraťte se na organizaci Free
21 * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java je obchodní značka nebo registrovaná obchodní značka
25 * Sun Microsystems, Inc.ve Spojených státech a jiných zemích.]
26 *
27 * -----
28 * MonthConstants.java
29 * -----
30 * (C) Copyright 2002, 2003, Object Refinery Limited.
31 *
32 * Původní autor: David Gilbert (pro Object Refinery Limited);
33 * Přispěvatelé: -;
34 *
35 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
36 *
```

```
37 * Změny
38 * -----
39 * 29. května 2002 : Verze 1 (kód je přemístěn ze třídy SerialDate) (DG);
40 *
41 */
42
43 package org.jfree.date;
44
45 /**
46 * Užitečné konstanty pro měsíce. Všimněte si, že nejsou ekvivalentní
47 * s konstantami definovanými v java.util.Calendar (kde leden=0 a prosinec=11).
48 * <P>
49 * Použito třídami SerialDate a RegularTimePeriod.
50 *
51 * @author David Gilbert
52 */
53 public interface MonthConstants {
54
55     /** Konstanta pro leden. */
56     public static final int JANUARY = 1;
57
58     /** Konstanta pro únor. */
59     public static final int FEBRUARY = 2;
60
61     /** Konstanta pro březen. */
62     public static final int MARCH = 3;
63
64     /** Konstanta pro duben. */
65     public static final int APRIL = 4;
66
67     /** Konstanta pro květen. */
68     public static final int MAY = 5;
69
70     /** Konstanta pro červen. */
71     public static final int JUNE = 6;
72
73     /** Konstanta pro červenec. */
74     public static final int JULY = 7;
75
76     /** Konstanta pro srpen. */
77     public static final int AUGUST = 8;
78
79     /** Konstanta pro září. */
80     public static final int SEPTEMBER = 9;
81
82     /** Konstanta pro říjen. */
83     public static final int OCTOBER = 10;
84
85     /** Konstanta pro listopad. */
86     public static final int NOVEMBER = 11;
87
88     /** Konstanta pro prosinec. */
89     public static final int DECEMBER = 12;
90
91 }
```

**Výpis B.4. 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("Invalid Month Code should throw exception");
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("Invalid month code should throw exception");
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("Hello"));
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(SEPTMBER, 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
281 SpreadsheetDate(day, month, year);}
282
283 public void testAddMonths() throws Exception {
284     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
288     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
289     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
290
291     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
292     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
293
294     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
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("Invalid day of week code should throw exception");
313 } catch (IllegalArgumentException e) {
314 }
315 }
316
317 public void testGetFollowingDayOfWeek() throws Exception {
318     // assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25,
319     DECEMBER, 2004)));
320     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26,
321     DECEMBER, 2004)));
322     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28,
323     FEBRUARY, 2004)));
324
325     try {
326         getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
327         fail("Invalid day of week code should throw exception");
328     } catch (IllegalArgumentException e) {
329 }
330
331 public void testGetNearestDayOfWeek() throws Exception {
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
334     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
335     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
337     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
338     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
339
340 //todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
343     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
344     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
345     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
346     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
347
348 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
349 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
352     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
353     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
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("Invalid day of week code should throw exception");
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("Invalid week code should throw exception");
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("Invalid relative code should throw exception");
432 //    } catch (IllegalArgumentException e) {
433 //    }
434 }
435
436 public void testCreateInstanceFromDDMMYYYY() 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),
451                 createInstance(new GregorianCalendar(1900,0,1).getTime()));
452     assertEquals(d(1, JANUARY, 2006),
453                 createInstance(new GregorianCalendar(2006,0,1).getTime()));
454 }
455 public static void main(String[] args) {
456     junit.textui.TestRunner.run(BobsSerialDateTest.class);
457 }
```

**Výpis B.5. SpreadsheetDate.java**

```
1 /* -----
2 * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
3 * -----
4 *
5 * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
6 *
7 * Informace o projektu: http://www.jfree.org/jcommon/index.html
8 *
9 * Tato knihovna je volný software; můžete ji šířit a modifikovat
10 * za podmínek GNU Lesser General Public License, jak byla publikovaná
11 * organizací Free Software Foundation; bud verze 2.1 licence, nebo
12 * (podle vaší volby) jakákoli pozdější verze.
13 *
14 * Tato knihovna je šířena a očekává se, že bude užitečná, ale
15 * BEZ JAKÉKOLIV ZÁRUKY; dokonce i bez implikované záruky z důvodu UVEDENÍ
16 * NA TRH nebo VHODNOSTI PRO URČITÝ ÚČEL. Viz GNU Lesser General Public
17 * License, kde naleznete další detaily.
18 *
19 * Kopii licence GNU Lesser General Public License byste měli obdržet
20 * spolu s touto knihovnou; pokud ne, obrátte se na organizaci Free
21 * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-
22 * 1301,
23 * USA.
24 * [Java je obchodní značka nebo registrovaná obchodní značka
25 * Sun Microsystems, Inc.ve Spojených státech a jiných zemích.]
26 *
27 * -----
28 * SpreadsheetDate.java
29 * -----
30 * (C) Copyright 2000-2005, Object Refinery Limited a jiní přispěvatelé.
31 *
32 * Původní autor: David Gilbert (pro Object Refinery Limited);
33 * Přispěvatelé: -;
34 *
35 * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36 *
37 * Změny
38 * -----
39 * 11-říjen-2001 : Verze 1 (DG);
40 * 05-listopad-2001 : Přidané metody getDescription() a setDescription() (DG);
41 * 12-listopad-2001 : Změna jména z ExcelDate.java na SpreadsheetDate.java
        (DG);
42 * Opravena chyba ve výpočtu dne, měsíce a roku ze sériového
43 * čísla (DG);
44 * 24-leden-2002 : Opravena chyba při ve výpočtu sériového čísla ze dne,
45 * měsíce a roku. Díky Trevoru Hillovi za oznámení (DG);
46 * 29-květen-2002 : Přidán objekt equals a metoda SourceForge, ID 558850 (DG);
47 * 03-říjen-2002 : Opraveny chyby, které oznámil Checkstyle (DG);
48 * 13-březen-2003 : Implementace třídy Serializable (DG);
49 * 04-září-2003 : Dokončeny metody isInRange() (DG);
```

```
50 * 05-září-2003 : Implementována třída Comparable (DG);
51 * 21-říjen-2003 : Přidána metoda hashCode() (DG);
52 *
53 */
54
55 package org.jfree.date;
56
57 import java.util.Calendar;
58 import java.util.Date;
59
60 /**
61 * Reprezentuje datum s použitím celého čísla, podobně jako
62 * implementace v Microsoft Excel. Rozsah podporovaných dat je od
63 * 1.ledna 1900 do 31.prosince 9999.
64 * <P>
65 * Pozor na záměrnou chybu v Excelu, která počítá rok
66 * 1900 jako přestupný, ten ale přestupný není. Více informací naleznete
67 * na internetových stránkách firmy Microsoft v článku Q181370:
68 * <P>
69 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70 * <P>
71 * Excel používá konvenci 1-Jan-1900 = 1. Tato třída používá
72 * konvenci 1-Jan-1900 = 2.
73 * Výsledkem je, že pro leden a únor 1900 bude v této třídě číslo dne
74 * jiné, než které Excel vypočítá...pak ale Excel přidá jeden den navíc
75 * (29-Feb-1900, které ale neexistuje!) A od tohoto dne
76 * se již čísla dnů budou shodovat.
77 *
78 * @author David Gilbert
79 */
80 public class SpreadsheetDate extends SerialDate {
81
82     /** Pro serializaci. */
83     private static final long serialVersionUID = -2039586705374454461L;
84
85     /**
86      * Číslo dne (1.leden 1900 = 2, 2.leden 1900 = 3, ..., 31.prosinec 9999 =
87      * 2958465).
88      */
89     private int serial;
90
91     /** Den v měsíci (1. až 28, 29, 30. nebo 31, v závislosti na měsíci). */
92     private int day;
93
94     /** Měsíc v roce (1 až 12). */
95     private int month;
96
97     /** Rok (1900 až 9999). */
98     private int year;
99
100    /** Volitelný popis pro datum. */
101    private String description;
```

```
102
103     /**
104      * Vytvoří novou instanci data.
105      *
106      * @param day den (v rozsahu od 1. do 28./29./30./31.).
107      * @param month měsíc (v rozsahu od 1 do 12).
108      * @param year rok (v rozsahu od 1900 do 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         // sériové číslo musí být synchronizováno se dnem-měsícem-rokem...
139         this.serial = calcSerial(day, month, year);
140
141         this.description = null;
142
143     }
144
145     /**
146      * Standardní konstruktor - vytvoří nový datový objekt reprezentující
147      * dané číslo dne (které by mělo být v rozsahu od 2 do 2958465).
148      *
149      * @param serial sériové číslo dne (rozsah: 2 až 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     // den-měsíc-rok je třeba synchronizovat se sériovým číslem...
162     calcDayMonthYear();
163
164 }
165
166 /**
167 * Vrací popis, který je připojen k datu. Není nutné,
168 * aby mělo datum popis, ale pro některé aplikace to je
169 * užitečné.
170 *
171 * @return Popis je připojen k datu.
172 */
173 public String getDescription() {
174     return this.description;
175 }
176
177 /**
178 * Nastavuje popis data.
179 *
180 * @param description popis tohoto data (hodnota <code>null</code>
181 * je povolena).
182 */
183 public void setDescription(final String description) {
184     this.description = description;
185 }
186
187 /**
188 * Vrací sériové číslo data, kde 1.ledna 1900 = 2
189 * (to odpovídá téměř vždy číselnému systému, který používá Microsoft
190 * Excel pro Windows a Lotus 1-2-3).
191 *
192 * @return sériové číslo tohoto data.
193 */
194 public int toSerial() {
195     return this.serial;
196 }
197
198 /**
199 * Vrací <code>java.util.Date</code>, což je ekvivalent tohoto data.
200 *
201 * @return Datum.
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 * Vrací rok (předpokládáme platný rozsah od 1900 do 9999).
211 *
212 * @return Rok.
213 */
214 public int getYYYY() {
215     return this.year;
216 }
217
218 /**
219 * Vrací měsíc (Leden = 1, Únor = 2, Březen = 3).
220 *
221 * @return Měsíc v roce.
222 */
223 public int getMonth() {
224     return this.month;
225 }
226
227 /**
228 * Vrací den v měsíci.
229 *
230 * @return Den v měsíci.
231 */
232 public int getDayOfMonth() {
233     return this.day;
234 }
235
236 /**
237 * Vrací kód reprezentující den v týdnu.
238 * <P>
239 * Kódy jsou definovány ve třídě {@link SerialDate} takto:
240 * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241 * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, a
242 * <code>SATURDAY</code>.
243 *
244 * @return Kód, reprezentující den v týdnu.
245 */
246 public int getDayOfWeek() {
247     return (this.serial + 6) % 7 + 1;
248 }
249
250 /**
251 * Testuje rovnost tohoto data s libovolným objektem.
252 * <P>
253 * Tato metoda vrátí hodnotu true JEN v případě, že objekt je instancí
254 * základní třídy {@link SerialDate}, a že reprezentuje stejný den jak tato
255 * třída {@link SpreadsheetDate}.
256 *
257 * @param object porovnávaný objekt (hodnota <code>null</code> je povolena).
```

```
258     *
259     * @return Logická proměnná.
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     * Vrací hešový kód této instance.
275     *
276     * @return Hash code.
277     */
278    public int hashCode() {
279        return toSerial();
280    }
281
282    /**
283     * Vrací rozdíl (ve dnech) mezi tímto datem a daným
284     * "jiným" datem.
285     *
286     * @param druhé datum, se kterým se provádí srovnání.
287     *
288     * @return Rozdíl (ve dnech) mezi tímto datem a daným
289     * "jiným" datem.
290     */
291    public int compare(final SerialDate other) {
292        return this.serial - other.toSerial();
293    }
294
295    /**
296     * Implementuje metodu, kterou požaduje rozhraní Comparable.
297     *
298     * @param other jiný objekt (obvykle jiná instance třídy SerialDate).
299     *
300     * @return Záporné celé číslo, nula nebo kladné celé číslo je v tomto
301     * objektu menší, rovno nebo větší než daný objekt.
302     */
303    public int compareTo(final Object other) {
304        return compare((SerialDate) other);
305    }
306
307    /**
308     * Vrací hodnotu true, jestliže aktuální instance reprezentuje stejné datum
309     * jako parametr.
```

```
310     *
311     * @param other datum, se kterým provádíme srovnání.
312     *
313     * @return vrací hodnotu <code>true</code>, pokud aktuální instance
314     * reprezentuje stejné datum jako parametr.
315     */
316    public boolean isOn(final SerialDate other) {
317        return (this.serial == other.toSerial());
318    }
319
320    /**
321     * Vrací hodnotu true, pokud aktuální instance reprezentuje dřívější
322     * datum než parametr.
323     *
324     * @param other datum, se kterým se provádí srovnání.
325     *
326     * @return vrací hodnotu <code>true</code>, pokud aktuální instance
327     * reprezentuje dřívější datum než parametr.
328     */
329    public boolean isBefore(final SerialDate other) {
330        return (this.serial < other.toSerial());
331    }
332
333    /**
334     * Vrací hodnotu true, pokud aktuální instance reprezentuje
335     * stejné datum jako parametr.
336     *
337     * @param other Datum, se kterým porovnáváme.
338     *
339     * @return vrací hodnotu <code>true</code>, pokud aktuální instance
340     * reprezentuje stejné datum jako parametr.
341     */
342    public boolean isOnOrBefore(final SerialDate other) {
343        return (this.serial <= other.toSerial());
344    }
345
346    /**
347     * Vrací hodnotu true, pokud aktuální instance reprezentuje stejné
348     * datum jako parametr.
349     *
350     * @param other datum, se kterým porovnáváme.
351     *
352     * @return vrací hodnotu <code>true</code>, pokud aktuální instance
353     * reprezentuje stejné datum jako parametr.
354     */
355    public boolean isAfter(final SerialDate other) {
356        return (this.serial > other.toSerial());
357    }
358
359    /**
360     * Vrací hodnotou true, pokud aktuální instance reprezentuje stejné
361     * datum jako parametr.
```

```
362     *
363     * @param other datum, se kterým porovnáváme.
364     *
365     * @return vrací kód <code>true</code>, pokud aktuální instance
366     * reprezentuje stejné datum jako parametr.
367     */
368     public boolean isOnOrAfter(final SerialDate other) {
369         return (this.serial >= other.toSerial());
370     }
371
372     /**
373      * Vrací hodnotu <code>true</code>, pokud aktuální instance {@link
374      * SerialDate} spadá do daného intervalu (VČETNĚ). Pořadí dat d1 a d2
375      * není důležité.
376      *
377      * @param d1 hraniční datum intervalu.
378      * @param d2 druhé hraniční datum intervalu.
379      *
380      * @return Logická hodnota.
381      */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383         return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
386     /**
387      * Vrací hodnotu true, pokud aktuální instance spadá do daného
388      * intervalu (volající objekt určuje, zda jsou okrajové body součástí
389      * intervalu či nikoliv. Pořadí dat d1 a d2 není důležité.
390      *
391      * @param d1 první hraniční datum intervalu.
392      * @param d2 druhé hraniční datum intervalu.
393      * @param obsahuje kód, který určuje, zda jsou okrajová data intervalu
394      * jeho součástí či ne.
395      *
396      * @return vrací hodnotu <code>true</code>, pokud aktuální instance
397      * spadá do daného intervalu.
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 * Spočítá sériové číslo ze dne, měsíce a roku.
423 * <P>
424 * 1. leden 1900 = 2.
425 *
426 * @param d den.
427 * @param m měsíc.
428 * @param y rok.
429 *
430 * @return sériové číslo dne měsíce a roku.
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 * Spočítá den, měsíc a rok ze sériového čísla.
446 */
447 private void calcDayMonthYear() {
448
449     // Dostaneme rok ze sériového data
450     final int days = this.serial - SERIAL_LOWER_BOUND;
451     // nadhodnoceno, protože jsme ignorovali přestupné roky
452     final int overestimatedYYYY = 1900 + (days / 365);
453     final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454     final int nonleapdays = days - leaps;
455     // Podhodnoceno, protože jsme nadhodnotili roky
456     int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458     if (underestimatedYYYY == overestimatedYYYY) {
459         this.year = underestimatedYYYY;
460     }
461     else {
462         int ssl = calcSerial(1, 1, underestimatedYYYY);
463         while (ssl <= this.serial) {
464             underestimatedYYYY = underestimatedYYYY + 1;
465             ssl = 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     // Obdržíme měsíc ze sériového data
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     // zůstává nám d(+1);
490     this.day = this.serial - ss2
491         - daysToEndOfPrecedingMonth[this.month] + 1;
492
493 }
494
495 }
```

#### Výpis B.6. RelativeDayOfWeekRule.java

```
1 /**
2  * =====
3  * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
4  * =====
5  * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
6  *
7  * Informace o projektu: http://www.jfree.org/jcommon/index.html
8  *
9  * Tato knihovna je volný software; můžete ji šířit a modifikovat
10 * za podmínek GNU Lesser General Public License, jak byla publikována
11 * organizací Free Software Foundation; buď verze 2.1 licence, nebo
12 * (podle vaší volby) jakákoliv pozdější verze.
13 *
14 * Tato knihovna je šířena a očekává se, že bude užitečná, ale
15 * BEZ JAKÉKOLIV ZÁRUKY; dokonce i bez implikované záruky z důvodu UVEDENÍ
16 * NA TRH nebo VHODNOSTI PRO URČITÝ ÚČEL. Viz GNU Lesser General Public
17 * License, kde najeznete další detaily.
18 *
19 * Kopii licence GNU Lesser General Public License byste měli obdržet
```

```
20 * spolu s touto knihovnou; pokud ne, obraťte se na organizaci Free
21 * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java je obchodní značka nebo registrovaná obchodní značka
25 * Sun Microsystems, Inc.ve Spojených státech a jiných zemích.]
26 *
27 * -----
28 * RelativeDayOfWeekRule.java
29 * -----
30 * (C) Copyright 2000-2003, Object Refinery Limited a spolupracovníci.
31 *
32 * Původní autor: David Gilbert (pro Object Refinery Limited);
33 * Přispěvatelé: -;
34 *
35 * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Změny (od 26.října 2001)
38 * -----
39 * 26.říjen 2001 : Změna balíčku na com.jrefinery.date.*;
40 * 03.říjen 2002 : Oprava chyb, které nahlásil Checkstyle (DG);
41 *
42 */
43
44 package org.jfree.date;
45
46 /**
47 * Pravidlo pro výpočet data v roce, které vrací datum v každém roce, na základě
48 * (a) referenčního pravidla; (b) dne v týdnu a (c) výběrového parametru
49 * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
50 * <P>
51 * Například Velký pátek lze definovat jako "Pátek PŘEDCHÁZEJÍcí
52 * Boží hod velikonoční".
53 *
54 * @author David Gilbert
55 */
56 public class RelativeDayOfWeekRule extends AnnualDateRule {
57
58     /** Odkaz na pravidlo výpočtu data, na kterém je tohle pravidlo založeno. */
59     private AnnualDateRule subrule;
60
61     /**
62      * Den v týdnu (SerialDate.MONDAY, SerialDate.TUESDAY, atd.).
63      */
64     private int dayOfWeek;
65
66     /** Specifikace dne v týdnu (PRECEDING, NEAREST nebo FOLLOWING). */
67     private int relative;
68
69     /**
70      * Implicitní konstruktor - vytvoří pravidlo pro pondělí následující po 1.
71      * lednu.
```

```
71     */
72     public RelativeDayOfWeekRule() {
73         this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
74     }
75
76     /**
77      * Standardní konstruktor - vytvoří pravidlo na základě předaného pravidla
78      * nižší úrovně.
79      * @param subrule pravidlo, které určuje referenční datum.
80      * @param dayOfWeek den v týdnu, relativní vůči referenčnímu datu.
81      * @param relative ukazuje, *který* den v týdnu (předchozí, nejbližší
82      * nebo následující).
83      */
84     public RelativeDayOfWeekRule(final AnnualDateRule subrule,
85         final int dayOfWeek, final int relative) {
86         this.subrule = subrule;
87         this.dayOfWeek = dayOfWeek;
88         this.relative = relative;
89     }
90
91     /**
92      * Vrací pravidlo nižší úrovně (nazývané rovněž referenční pravidlo).
93      *
94      * @return Pravidlo pro výpočet data v roce, určující jeho
95      * referenční datum.
96      */
97     public AnnualDateRule getSubrule() {
98         return this.subrule;
99     }
100
101    /**
102     * Nastavuje referenční pravidlo.
103     *
104     * @param subrule Pravidlo pro roční výpočet data, určující jeho
105     * referenční datum.
106     */
107    public void setSubrule(final AnnualDateRule subrule) {
108        this.subrule = subrule;
109    }
110
111    /**
112     * Vrací den v týdnu podle tohoto pravidla.
113     *
114     * @return den v týdnu tohoto pravidla.
115     */
116    public int getDayOfWeek() {
117        return this.dayOfWeek;
118    }
119
120    /**
121     * Stanovuje den v týdnu pro toto pravidlo.
122     *
```

```
123     * @param dayOfWeek den v týdnu (SerialDate.MONDAY,  
124     * SerialDate.TUESDAY, atd.).  
125     */  
126     public void setDayOfWeek(final int dayOfWeek) {  
127         this.dayOfWeek = dayOfWeek;  
128     }  
129  
130     /**  
131      * Vrací "relativní" atribut, který určuje, *jaký*  
132      * den v týdnu nás zajímá (SerialDate.PRECEDING,  
133      * SerialDate.NEAREST nebo SerialDate.FOLLOWING).  
134      *  
135      * @return "relativní" atribut.  
136      */  
137     public int getRelative() {  
138         return this.relative;  
139     }  
140  
141     /**  
142      * Nastavuje "relativní" atribut (SerialDate.PRECEDING, SerialDate.NEAREST,  
143      * SerialDate.FOLLOWING).  
144      *  
145      * @param relative určuje *který* den v týdnu je tímto pravidlem  
146      * vybrán.  
147      */  
148     public void setRelative(final int relative) {  
149         this.relative = relative;  
150     }  
151  
152     /**  
153      * Vytvoří klon tohoto pravidla.  
154      *  
155      * @return vrací klon tohoto pravidla.  
156      *  
157      * @throws CloneNotSupportedException K tomu by nikdy nemělo dojít.  
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      * Vrací datum generované tímto pravidlem pro daný rok.  
168      *  
169      * @param year rok (1900 <= year <= 9999).  
170      *  
171      * @return Datum generované pravidlem pro daný rok (je možná  
172      * i hodnota <code>null</code>).  
173      */  
174     public SerialDate getDate(final int year) {
```

```
175     // zkontroluj argument...
176     if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
177         || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
178         throw new IllegalArgumentException(
179             "RelativeDayOfWeekRule.getDate(): year outside valid range.");
180     }
181
182
183     // vypočítej datum...
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 }
```

**Výpis B.7. DayDate.java (poslední verze)**

```
1 /**
2 * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
3 *
4 *
5 * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
...
36 */
37 package org.jfree.date;
38
39 import java.io.Serializable;
40 import java.util.*;
41
42 /**
43 * Abstraktní třída reprezentující neměnná data s přesností
```

```
44 * jednoho dne. Implementace přiřazuje každému datu celé číslo, které
45 * reprezentuje pořadové číslo dne nějakého fixního původu.
46 *
47 * Proč prostě nepoužívat java.util.Date? Bude-li to bude mít smysl, pak ano.
48 * Prozatím balíček java.util.Date může být *příliš* přesný - je to v tomto
49 * okamžiku přesnost na 1/1000 sekundy (se samotným datem, které závisí
50 * na časovém pásmu). Někdy jen chceme znázornit konkrétní den (např. 21
51 * ledna 2015), aniž bychom se zabývali časem nebo časovým pásmem
52 * nebo čimkoliv jiným. To je důvod, proč jsme vytvořili DayDate.
53 *
54 * Pro vytvoření instance použijte DayDateFactory.makeDate.
55 *
56 * @author David Gilbert
57 * @author Robert C. Martin provedl značnou část refaktorování.
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 }
```

#### Výpis B.8. Month.java (poslední verze)

```
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    private int index;
14
15    Month(int index) {
16        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("Invalid month index " + 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("Invalid month " + 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 }
```

**Výpis B.9. Day.java (poslední verze)**

```
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("Illegal day index: %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 is not a valid weekday string", s));
45    }
46
47    public String toString() {
48        return dateSymbols.getWeekdays()[index];
49    }
50
51    public int toInt() {
52        return index;
53    }
54 }
```

**Výpis B.10. DateInterval.java (poslední verze)**

```
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 }
```

**Výpis B.11. WeekInMonth.java (poslední verze)**

```
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 }
```

**Výpis B.12. WeekdayRange.java (poslední verze)**

```
1 package org.jfree.date;
2
3 public enum WeekdayRange {
4     LAST, NEAREST, NEXT
5 }
```

**Výpis B.13. DateUtil.java (poslední verze)**

```
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 - 1996) / 4;
28        int leap100 = (year - 1800) / 100;
29        int leap400 = (year - 1600) / 400;
30        return leap4 - leap100 + leap400;
31    }
32 }
```

**Výpis B.14. DayDateFactory.java (poslední verze)**

```
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);
10    protected abstract DayDate _makeDate(int day, Month month, int year);
11    protected abstract DayDate _makeDate(int day, int month, int year);
12    protected abstract DayDate _makeDate(java.util.Date date);
13    protected abstract int _getMinimumYear();
14    protected abstract int _getMaximumYear();
15
16    public static DayDate makeDate(int ordinal) {
```

```
17     return factory._makeDate(ordinal);
18 }
19
20 public static DayDate makeDate(int day, Month month, int year) {
21     return factory._makeDate(day, month, year);
22 }
23
24 public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26 }
27
28 public static DayDate makeDate(java.util.Date date) {
29     return factory._makeDate(date);
30 }
31
32 public static int getMinimumYear() {
33     return factory._getMinimumYear();
34 }
35
36 public static int getMaximumYear() {
37     return factory._getMaximumYear();
38 }
39 }
```

**Výpis B.15. SpreadsheetDateFactory.java (poslední verze)**

```
1 package org.jfree.date;
2
3 import java.util.*;
4
5 public class SpreadsheetDateFactory extends DayDateFactory {
6     public DayDate _makeDate(int ordinal) {
7         return new SpreadsheetDate(ordinal);
8     }
9
10    public DayDate _makeDate(int day, Month month, int year) {
11        return new SpreadsheetDate(day, month, year);
12    }
13
14    public DayDate _makeDate(int day, int month, int year) {
15        return new SpreadsheetDate(day, month, year);
16    }
17
18    public DayDate _makeDate(Date date) {
19        final GregorianCalendar calendar = new GregorianCalendar();
20        calendar.setTime(date);
21        return new SpreadsheetDate(
22            calendar.get(Calendar.DATE),
23            Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24            calendar.get(Calendar.YEAR));
25    }
26}
```

```
27     protected int _getMinimumYear() {
28         return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29     }
30
31     protected int _getMaximumYear() {
32         return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33     }
34 }
```

**Výpis B.16. SpreadsheetDate.java (poslední verze)**

```
1 /**
2  * JCommon : volná knihovní třída pro všeobecné použití na platformě Java
3  *
4  *
5  * (C) Copyright 2000-2005, Object Refinery Limited and Contributors.
6  *
7  * ...
8  */
9 package org.jfree.date;
10
11 /**
12  * Reprezentuje datum ve formátu celého čísla. Způsob je podobný tomu, jak
13  * implementuje datum Microsoft Excel. Rozsah dat je podporován od
14  * 1.ledna 1900 do 31.prosince 9999.
15  * <p>
16  * Pozor na záměrnou chybu v Excelu, která počítá rok
17  * 1900 jako přestupný, ten ale přestupný není. Více informací naleznete
18  * na internetových stránkách firmy Microsoft v článku Q181370:
19  * <p>
20  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
21  * <p>
22  * Excel používá konvenci 1-Jan-1900 = 1. Tato třída používá
23  * konvenci 1-Jan-1900 = 2.
24  * Výsledek je, že v této třídě bude číslo dne jiné než které
25  * Excel vypočítá pro leden a únor 1900...pak ale Excel přidá jeden den navíc
26  * (29-Feb-1900, které ale neexistuje!) Od tohoto dne
27  * se již budou čísla dnů shodovat.
28  *
29  * @author David Gilbert
30  */
31 public class SpreadsheetDate extends DayDate {
32     public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
33     public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
34     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
35     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                 "The 'year' argument must be in range " +
100                 MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101         if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102             throw new IllegalArgumentException("Invalid 'day' argument.");
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: Serial must be in range 2 to 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 }
```

# Doslov

Když jsem se v roce 2005 zúčastnil konference na téma agilní programování v Denveru, předala mi Elizabeth Hedricksonová<sup>1</sup> zelený náramek. Byl to ten druh, který tak zpopularizoval Lance Armstrong. Tento měl nápis „Posedlý testy“. Rád jsem si ho nasadil a nosil jsem ho s pýchou. Od počátku svého studia TDD (vývoje řízeného testy) od Kenta Becka v roce 1999 jsem byl touto metodou opravdu posedlý. Ale pak se stalo něco zvláštního. Nemohl jsem tento náramek sundat. Ne z důvodů fyzických, ale z důvodů *morálních*. Náramek sděloval otevřeně cosi o mé profesionální etice. Byl to viditelný indikátor mého závazku psát ten nejlepší kód, jakého jsem schopen. Sundat ho mi připadalo jako zrada této etiky a tohoto závazku.

Takže ho na zápěstí stále nosím. Když píšu nějaký kód, mám jej ve svém periferním zorném poli. Neustále mi připomíná můj slib, který jsem si dal, že budu psát čistý kód.

---

1. <http://www.qualitytree.com/>.



# Rejstřík

## A

- abstrakce, 296, 315
- addDays, 279, 284
- afinita, konceptuální, 103
- algoritmus, pochopení, 302
- analyzátor, argumentů, 204
- antisymetrie
  - datová, 113
  - objektová, 113
- AOP, 174
- Args, 249–250, 257
  - třída, 204
- ArgsException, 247, 249, 257
- argument
  - funkce, 61
  - logický, 62, 294
  - přepínací, 299
  - řetězcový, 222
  - seznam, 64
  - výstupní, 65, 294
- ArgumentMarshaler, 223–225
- architektura
  - systémová, 179
  - testování, 179
- AspectJ, 179
- aspektově orientované programování, 174
- automatizované, kódování, 200

## B

- BDUF (Big Design Up Front), 179
- Bjarne Stroustrup, 30
- blok, 57
- BobsSerialDateTest.java, 380
- BooleanArgumentMarshaler, 224–226, 240, 243

## C

- CAS (Compare and Swap), 331
  - cena, celková, 27
  - ClassCastException, 232
  - compactString, 270
  - ComparisonCompactor, 260, 262
  - computeCommonSuffix, 268–269
  - currentArgs, 239
- ## Č
- čekání, cyklické, 340–341
  - číslo, magické, 305
  - členská předpona, 45
  - čtenáři-zapisovatelé, 195
  - čtení, kódu, 58

## D

- data
  - konfigurační, 310
  - kopie, 193
  - přenos, 117
  - rozsah, 193
- DateInterval.java, 408
- DateUtil.java, 409
- datová abstrakce, 112
- Dave Thomas, 32
- Day.java, 407
- DayDate, 277, 284
- DayDate.java, 402
- DayDateFactory.java, 409
- dědění, konstant, 312
- definice
  - normálního toku, 127
  - tříd výjimek, 125
- deklarace, proměnných, 100
- Démétřin zákon, 115

dezinformace, 41  
DI (Dependency Injection), 170, 184  
DIP (Dependency Inversion Principle), 165  
doménově specifický jazyk, 181  
dotaz, oddělování, 66  
DoubleArgumentMarshaler, 245  
DSL (Domain-Specific Languages), 181  
DTO (data transfer object), 117

**E**

Eclipse, 32  
efekt, vedlejší, 65, 317  
Error.java, 68  
ErrorCode, 246  
errorMessage, 257  
expresivita, 187  
Extreme Programming Adventures in C#, 33  
Extreme Programming Installed, 33  
eXtémní Programování, 34

**F**

F.I.R.S.T., 149  
fast, 149  
findCommonPrefix, 268  
findCommonPrefixAndSuffix, 268  
findCommonSuffix, 267–268  
Fit, 34  
formatCompactedComparison, 266, 272  
formátování, 96  
– důvody, 96  
– horizontální, 104  
– vertikální, 96  
funkce, 54, 86, 293, 307  
– argumenty, 61  
– malá, 56  
– mrtvá, 294  
– názvy, 302  
– příliš mnoho argumentů, 293  
– s jedním argumentem, 62  
– se dvěma argumenty, 63  
– se třemi argumenty, 63

– sekce, 58  
– sestup, 309  
– tvary, 62  
– úroveň abstrakce, 58  
– záhlaví, 90  
– závislá, 102  
funkčnost, nekorektní, 295

**G**

getBoolean, 226, 232  
getFollowingDayOfWeek, 275, 279  
getMonths, 283  
getNearestDayOfWeek, 275, 286  
getPreviousDayOfWeek, 275, 285  
Grady Booch, 31

**H**

hodnota, null, 128  
horizontální formátování, 104  
horizontální hustota, 104  
horizontální oddělování, 104  
horizontální zarovnání, 105  
hranice, 134  
– čistá, 137  
Hungarian Notation, 45  
hustota  
– horizontální, 104  
– vertikální, 99  
hybrid, 116

**CH**

chyba  
– programová, 318  
– zpracování, 68

**I**

if/else, 304  
implementace, 46  
Implementation Patterns, 26  
independent, 149  
informace, nelokální, 89  
instanční proměnná, 101

**IntegerArgumentMarshaler**, 227, 244

**IoC** (Inversion of Control), 170

**isIntArg**, 229

**isStringArg**, 229

**izolování**, od změn, 164

## J

**Java**, 311

**Javadoc**, 79

- v nevěřejném kódu, 90

**jazyk**, testovací, 145

**jméno**, 314

- dlouhé, 317

- domén problému, 49

- jednoznačné, 316

- kódování, 45, 317

- metod, 47

- popisné, 61, 314

- skryté překládání, 46

- tříd, 47

- výběr, 315

- vyhledání, 44

- vyslovitelné, 43

- vysvětlující význam, 40

- z domény řešení, 48

**JUnit**, 260

## K

**klíčové slovo**, 64

**klient/server**, 322, 345

- s podprocesy, 348

**kód**

- bez podprocesů, 198

- čistý, 29–30

- čtení, 58

- chybový, 66

- korektní, 197

- mrtvý, 297

- návratový, 122

- počet cest, 325

- pokrytí, 319

- pro vypínání, 197

- procedurální, 115

- provádění, 325

- souběžný, 198

- špatný, 26

- testování, 342

- třetích stran, 132

- vyjádření, 75

- zakomentovaný, 87, 293

- zánik, 26

- zdvojený, 185

**kódování**

- automatizované, 200

- jmen, 45, 317

- ruční, 200

**kolekce**, 194

**komentář**, 74, 86, 292

- deníkový, 82

- dobrý, 75

- informativní, 76

- matoucí, 82

- na konci složených závorek, 86

- nadbytečný, 80, 292

- nejasná spojitost, 89

- nevhodný, 292

- obsahující šum, 83

- právnického charakteru, 75

- připisování, 87

- špatně napsaný, 293

- špatný, 79

- TODO, 78

- ve formátu HTML, 88

- zastaralý, 292

- závazný, 82

**konceptuální, afinita**, 103

**konstanta**, 313

- dědění, 312

- pojmenovaná, 305

**kontext**, 50

- s výjimkami, 125

konvence, 306  
– standardní, 304

## L

lastDayOfMonth, 284  
leapYearCount, 284  
log4j, 134

## M

magnet, závislosti, 68  
metoda  
– jméno, 47  
– minimální, 188  
– synchronizovaná, 196  
Michael Feathers, 32  
model, běhový, 194  
modifikátor, static, 301  
modul, zásuvný, 198  
Monte Carlo, 344  
Month.java, 405  
monthCodeToQuarter, 278, 283  
monthCodeToString, 283  
MonthConstants, 277  
MonthConstants.java, 378

## N

nástroj, pro krytí, 318  
návratový, kód, 122  
návrh  
– pravidla, 184  
– vyvíjející se, 184  
návrhový vzor, 34  
název, funkce, 302  
názvosloví, standardní, 316  
nekonzistentnost, 298  
normální tok, 127  
notace, maďarská, 45  
null, 128  
– předávání, 129  
NumberFormatException, 227

## O

objasnění, 77  
Object Oriented Analysis and Design with Applications, 31  
objekt  
– jako argument, 64  
– pro přenos dat, 117  
objektově orientované programování, 34  
obor, prázdný, 108  
OCP (Open Closed Principle), 60  
oddělování  
– dotazů, 66  
– horizontální, 104  
– pojmu, 98  
– příkazů, 66  
– vertikální, 298  
odkaz, tranzitivní, 311  
odpovědnost, špatně umístěná, 301  
odsazování, 57, 106  
– nedodržování pravidel, 107  
opakování, 68  
optimalizace, rozhodování, 180  
org.jfree.date.SerialDate, 351  
organizace, podporující změny, 162

## P

paradox, prvotní, 29  
počet, cest, 326  
podmínka  
– hraniční, 309, 318  
– zapouzdření, 306, 309  
podproces  
– nezávislý, 193  
– problémy, 198  
– přidání, 323  
podtitulek, se jmény, 87  
pojem, vertikální oddělování, 98  
polymorfismus, 304  
pozice, označení, 86  
pravidla  
– formátovací, 109

– týmová, 108  
pravidlo, skautské, 36  
princip, jedné odpovědnosti, 154  
principy, 37  
producent-spotřebitel, 195  
program, výukový, 345  
programování, strukturované, 69  
proměnná, 86  
– deklarace, 100  
– instanční, 101  
– vysvětlující, 301  
propustnost  
– kalkulace, 337–338  
– zvyšování, 336  
prostředí, 293  
průnik, zájmů, 173  
předávání, hodnoty null, 129  
předpona, členská, 45  
prekládání  
– jmen, 46  
– skryté, 46  
přesnost, 306  
příkaz, oddělování, 66  
příkazový řádek, 204  
přístup, 28

**R**

rámeč  
– běhový, 330  
– čistý, 176  
refaktorování, 185  
– trídy `SerialDate`, 274  
rekonstrukce, celková, 28  
`RelativeDayOfWeekRule.java`, 398  
`relativeToString`, 276  
`repeatable`, 149  
Ron Jeffries, 33  
rozbor, analyzátoru argumentů, 204  
rozdíl, smysluplný, 42  
rozhodování, optimalizace, 180  
rozhraní, 46

rozsah, platnosti identifikátorů, 123  
ruční kódování, 200

**R**

řešení, bez blokace, 330

**S**

sekce, synchronizovaná, 196  
`self-validating`, 149  
selhání, 199  
– nejasné, 198  
– tolerování, 334  
selhávání, zákonitosti, 318  
separování, modulu Main, 169  
`SerialDate`, 274, 276  
`SerialDate.Java`, 352  
`SerialDateTest.java`, 372  
`SerialDateTests`, 274  
server, 322  
– modifikovaný, 196  
– sledování, 324  
sestavení, 293  
`setArgument`, 231, 238, 240  
`setBoolean`, 224–225  
`setBooleanArg`, 239–242  
`setGet`, 226  
`setIntArg`, 238–239, 241  
`setStringArg`, 241  
seznam  
– argumentů, 64  
– dlouhý, 311  
schopnost, chybějící, 298  
skautské pravidlo, 264  
sledování, serveru, 324  
sloveso, 64  
slovní hříčka, 48  
slovo, klíčové, 64  
`Smalltalk`, 34  
směr, myšlenkový, 34  
souběžnost, 190, 322  
– mýty, 191

- princip ochrany, 192
  - problémy, 192
  - soubor, více jazyků, 294
  - soudržnost, 156
  - souvislost, smysluplná, 49
  - SpreadSheetDate, 279
  - SpreadsheetDate.java, 389, 411
  - SpreadsheetDateFactory.java, 410
  - SRP (Single Responsibility Principle), 60, 184, 188, 192
  - standardy, 180
  - StringArgumentMarshaler, 225
  - stringToWeekdayCode, 281
  - struktura, 306
    - skrytá, 117
  - struktuované programování, 69
  - Switch, 59
  - switch/case, 304
  - systém, tvorba, 168
- Š**
- šablona, JUnit, 260
  - škálování, 171
- T**
- test, 293, 318
    - aktuální, 149
    - aserce, 147
    - čistý, 141–142
    - důkladný, 318
    - hraniční, 136
    - hraničních podmínek, 318
    - jednotkový, 140
    - kódu podprocesů, 197
    - kódu s podprocesy, 344
    - kódu s více podprocesy, 342
    - metodou Monte Carlo, 344
    - myšlenka, 148
    - nedostatečný, 318
    - nejednoznačný, 318
    - nezávislý, 149
- U**
- usporádání, vertikální, 103
  - úvod, 37
- V**
- varování, 78
  - vazba
    - skrytá časová, 307
    - umělá, 298
  - vertikální hustota, 99
  - vertikální usporádání, 103
  - vertikální vzdálenost, 99
  - vracení, chybových kódů, 66
  - výčet, 313

výjimka, 66, 122  
– definice tříd, 125  
– nekontrolovaná, 124  
vyloučení, vzájemné, 339–340  
výměna, nucená, 340  
výpočet, možných řazení, 326  
výraz, podmíněný, 306  
vývoj, řízený testy, 140  
vzdálenost, vertikální, 99

## W

Ward Cunningham, 34  
`WeekdayRange.java`, 408  
`WeekInMonth`, 286  
`WeekInMonth.java`, 408  
`weekInMonthToString`, 276, 286  
Wiki, 34  
*Working Effectively with Legacy Code*, 32

## Z

zabezpečení, zrušené, 295  
zablokování, 338  
zájem, průnik, 173  
zákon, Démétrin, 115  
záměr, nejasný, 300

zamykání, 196, 340  
– na straně klienta, 334  
– na straně serveru, 335  
zaneřáděnost, 298  
zapouzdření, 152  
– hraniční podmínky, 309  
– podmínky, 306  
zarovnání, horizontální, 105  
závislost  
– fyzická, 303  
– logická, 303  
– magnet, 68  
– mezi metodami, 33  
– mezi synchronizovanými metodami, 196  
– vkládání, 170  
záZNAM, aktivní, 118  
zdroj, získávání, 341  
zdvojení, 295  
změna, postupná, 220  
znak, zástupný, 311  
zpracování, chyb, 68  
zprostředkovatel, 174  
zvýraznění, 79

## Praktické příklady bez zbytečné teorie!

Edice Hotová řešení přináší základní stavební kameny pro vaše aplikace a řešení typických programátorských či administrátorských problémů. Kromě fragmentů předmětného programového kódu, skriptů nebo postupů obsahují jeho důkladná vysvětlení i potřebné vstupy do jednotlivých tématických okruhů. Kromě základních úloh a řešených problémů přidávají nejlepší autoři řadu dalších **inspirativních příkladů, efektivních postupů a zajímavých tipů** – jak pro vaše poznání dané technologie či jazyka, tak přímo pro vaše aplikace!



**Vše potřebné:** vývojové nástroje, zdrojový kód nebo skripty řešení ke snadné modifikaci, ale i užitečné nástroje a programy, vždy najdete na doprovodném CD.

### CSS

HOTOVÁ ŘEŠENÍ

Petr Staniček



268 stran + CD  
289 Kč/369 Sk  
prodejný kód K0994

### AJAX

HOTOVÁ ŘEŠENÍ

Luboslav Lacko

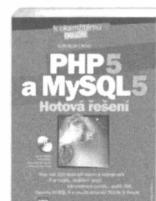


272 stran + DVD  
297 Kč/407 Sk  
prodejný kód K1522

### PHP A MYSQL 5

HOTOVÁ ŘEŠENÍ

2. AKTUALIZOVANÉ VYDÁNÍ  
Luboslav Lacko

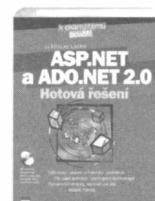


320 stran + CD  
349 Kč/449 Sk  
prodejný kód K1479

### ASP.NET

A ADO.NET 2.0

HOTOVÁ ŘEŠENÍ  
Luboslav Lacko



388 stran + CD  
369 Kč/517 Sk  
prodejný kód K1194

### SQL

HOTOVÁ ŘEŠENÍ

Luboslav Lacko



296 stran + CD  
297 Kč/449 Sk  
prodejný kód K0848

### C#

HOTOVÁ ŘEŠENÍ

Miroslav Virius



344 stran + CD  
397 Kč/549 Sk  
prodejný kód K1236

### VIUAL BASIC.NET

HOTOVÁ ŘEŠENÍ

Martin Görtler, Pavel Kocich



312 stran + CD  
289 Kč/429 Sk  
prodejný kód K1017

### AUTOCAD A AUTOCAD LT

HOTOVÁ ŘEŠENÍ

Peter Janeček

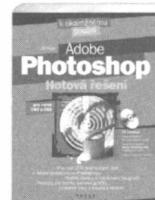


240 stran + CD  
289 Kč/399 Sk  
prodejný kód K1278

### ADOBE PHOTOSHOP

HOTOVÁ ŘEŠENÍ

Jiří Fotr

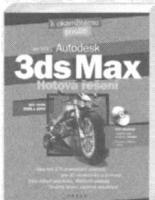


256 stran + CD  
297 Kč/407 Sk  
prodejný kód K1315

### 3DS MAX

HOTOVÁ ŘEŠENÍ

2. AKTUALIZOVANÉ VYDÁNÍ  
Jan Kříž



328 stran + DVD  
349 Kč/479 Sk  
prodejný kód K1623

### MICROSOFT WINDOWS SERVER 2003

HOTOVÁ ŘEŠENÍ

Patrik Malina



360 stran + DVD  
369 Kč/517 Sk  
prodejný kód K1310

### MICROSOFT EXCHANGE SERVER 2003

HOTOVÁ ŘEŠENÍ

Petr Štefka



328 stran + CD  
497 Kč/699 Sk  
prodejný kód K1230

### MICROSOFT OFFICE EXCEL 2007

HOTOVÁ ŘEŠENÍ

Josef Pecinovský

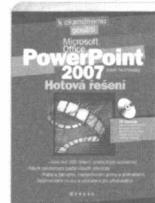


248 stran + CD  
249 Kč/339 Sk  
prodejný kód K1307

### MICROSOFT POWERPOINT 2007

HOTOVÁ ŘEŠENÍ

Josef Pecinovský



240 stran + CD  
269 Kč/369 Sk  
prodejný kód K1616

### MICROSOFT OFFICE PROJECT

HOTOVÁ ŘEŠENÍ

Karel Hyndrák



312 stran + CD  
319 Kč/449 Sk  
prodejný kód K1428